EX LIBRIS
UNIVERSITATIS
ALBERTENSIS

The Bruce Peel
Special Collections
Library

Digitized by the Internet Archive
in 2025 with funding from
University of Alberta Library

https://archive.org/details/0162018299675

University of Alberta

Library Release Form

Name of Author: Zhuang Guo

Title of Thesis: Developing Network Server Applications Using Generative Design Patterns

Degree: Master of Science

Year of Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's permission.

University of Alberta

Developing Network Server Applications Using Generative Design Patterns

By

Zhuang Guo

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta

Fall 2003

University of Alberta

Faculty of Graduate Study and Research

The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Developing Network Server Applications Using Generative Design Patterns** submitted by **Zhuang Guo** in partial fulfillment of the requirements for the degree of **Master of Science**.

Abstract

Design patterns are generic solutions to recurring software design problems that can be customized for particular applications. The Correct Object-Oriented Pattern-based Parallel Programming System (CO_2P_3S) uses design pattern templates to generate code for design patterns. The user chooses a design pattern template and customizes it by selecting a set of design options applicable to the application. Together with application-specific code, CO_2P_3S can generate a complete application. Prior to this thesis, CO_2P_3S has been used to generate small parallel and sequential applications. In contrast, this research evaluates the utility and performance of CO_2P_3S on larger applications. A generative design pattern template called the Network Server (N-Server) is constructed. It is shown that this pattern template can greatly ease the complexities involved in the construction of network server applications. The N-Server employs an event-driven approach as the concurrency model and uses the Reactor design pattern as the fundamental mechanism for event demultiplexing and dispatching. Many design options are offered, the existence of which makes the N-Server highly reconfigurable and therefore suitable for the construction of a large variety of network server applications, ranging from trivial cases like a Time server to those as sophisticated and performance-sensitive as an HTTP server. Based on the N-Server, two server applications, an HTTP server (COPS-HTTP) and an FTP server (COPS-FTP) are constructed. Only a small amount of effort is needed to develop them, since a large fraction of the code is generated using the N-Server. In terms of application performance, experiments are conducted to compare the performance of COPS-HTTP under static workloads against Apache, the most widely used Web server. It is found that COPS-HTTP generally outperforms

Apache while maintaining a high degree of service fairness. Two additional experiments were conducted to demonstrate the effectiveness of the event scheduling and automatic overload control mechanism of the N-Server. In conclusion, this research has successfully applied CO_2P_3S to the construction of network server applications and demonstrated the effectiveness of its code generation approach for larger applications.

Contents

1. Introduction **1**

1.1 Motivation 1

1.2 The scope of this thesis 4

1.3 Dissertation organization 5

2.CO_2P_3S Overview **7**

2.1 Introduction 7

2.2 Background 8

2.3 Design Pattern Templates 10

2.4 Development Process and Design Philosophy 11

2.5 MetaCO_2P_3S 15

3. Concurrency **20**

3.1 Concurrency Models for Server 20

3.2 Examples of Event-Driven Concurrency Models 23

3.2.1 Single-Process-Event-Driven 23

3.2.2 Multi-Process-Event-Driven 23

3.2.3 Staged-Event-Driven-Architecture 24

3.3 Summary 26

4. Network Server Application Design Patterns **27**

4.1 Reactor 27

4.2 Proactor 29

4.3 Acceptor-Connector 30

4.4 Asynchronous Completion Token 32

5. The Network Server Pattern **33**

5.1 Intent 33

5.2 Motivation 33

5.3 Applicability 38

5.4 Template Options 38

5.5 Architecture of Generated Code 42
5.6 Example 56

6. Applications 58
6.1 COPS-HTTP 58
6.2 COPS-FTP 60
6.3 Summary 63

7. Performance Evaluation 64
7.1 The Performance Experiment 64
7.2 The Event Scheduling Experiment 70
7.3 The Overload Control Experiment 72
7.4 Summary 73

8. Summary and Conclusions 75
8.1 Conclusions 75
8.2 Future Work 76

Bibliography 78

List of Tables

Table 5.1: List of Events 43
Table 5.2: Eight concurrency models 52
Table 5.3: Option selections vs. architecture changes 55
Table 6.1: The code distribution of COPS-HTTP 60
Table 6.2: The code distribution of COPS-FTP 62

List of Figures

Figure 2.1: Selecting design pattern templates 11
Figure 2.2: Setting options 11
Figure 2.3: Setting a lexical option 12
Figure 2.4: Setting a Design Option 12
Figure 2.5: Insert hook methods 13
Figure 2.6: Compilation 14
Figure 2.7: Execution 14
Figure 2.8: The Meta CO_2P_3S GUI 16
Figure 2.9: Template constants 16
Figure 2.10: Template classes 17
Figure 2.11: Template options 17
Figure 2.12: Template GUI configurations 18
Figure 3.1: Event-driven application layers 22
Figure 3.2: A staged-event-driven HTTP server 24
Figure 4.1: Reactor 28
Figure 4.2: Interaction diagram of the Proactor design pattern 29
Figure 5.1: Five basic steps 35
Figure 5.2: No decoder encoder 40
Figure 5.3: A centralized logging service 42
Figure 5.4: Event, Handle, and Event Handler 45
Figure 5.5: Event Sources 46
Figure 6.1: COPS-HTTP architecture 59
Figure 6.2: Storing a file 61
Figure 7.1: Experiment environment 66
Figure 7.2: Web server throughputs 67
Figure 7.3: Service fairness 68
Figure 7.4: Throughput linear models 69
Figure 7.5: Event scheduling experiment environment 70
Figure 7.6: Corporate portals vs. homepages 71

Figure 7.7: Response time..... 72

Chapter 1

Introduction

1.1 Motivation

Network server applications play a key role in networked computation, because they allow the sharing of critical system resources and provide services to multiple clients simultaneously. Almost all computer users in their daily activities explicitly or implicitly make use of several prominent network server applications, such as Web Server, Proxy Server, Ftp Server, Mail Server, Domain Name Services, etc. Although the World Wide Web has dramatically affected many aspects in today's society, the evolution and influence of computer networks are still at an early stage. With the growing availability and usage of computer networks, the demand for more heterogeneous network server applications will inevitably become greater.

However, building network server applications is essentially a difficult and time-consuming task due to the presence of two major challenges:

(1) Managing complexities embedded in the network-programming domain.

(2) Designing and implementing a concurrency strategy to effectively achieve the desired reliability and performance requirements.

The former of the two challenges is caused by the fact that network server applications must communicate with their peers and network programming is quite error-prone. The complexities associated with network communication are further grouped into two categories in [1]: inherent complexities and accidental complexities. Inherent complexities refer to requirements imposed on the communication software such as dealing with network and host failures, improving throughput and minimizing latencies, etc. Accidental complexities are mainly caused by limitations with using conventional tools and techniques in communication software development. For instance, to program with the low-level socket API, users must deal with many types of error conditions returned by function calls, which is both cumbersome and error-prone. On the other hand,

high-level communication mechanisms (like Java RMI, CORBA, and DCOM) lack key features such as asynchronous I/O and end-to-end Quality of Service support.

A concurrency strategy is necessary, since network server applications must serve multiple requests arriving simultaneously from clients and achieve application speedup and scaleup. Unfortunately, concurrent applications are in general much harder to implement and debug due to concerns related to inter-process communication and synchronization. Although traditional concurrency models based on processes or threads are relatively straightforward to implement, there are two serious drawbacks: they do not scale up very well and they have difficulty supporting request scheduling, because most popular operating systems are mainly designed to support time-sharing among processes or threads. To overcome these drawbacks, users often resort to event-driven concurrency models. However, these models are more difficult and time-consuming to implement.

In the face of challenges similar to those associated with the development of network server applications, the software industry often relies on expert knowledge and experiences to look for solutions. Design patterns [2], which capture such expert knowledge and experiences, are proven solutions to recurring problems in software design. With design patterns, junior developers can learn from their seniors' best practices and avoid making the same mistakes again. In order to construct network server applications effectively and efficiently, many design patterns have been discovered. For example in [3], sixteen design patterns were documented, which cover four key areas of the network server application domain: service access and configuration, event handling, synchronization, and concurrency. Further, the pioneering work of applying design patterns to the construction of high performance web servers has demonstrated the power of design patterns [1].

Although design patterns help to capture expert experiences, and therefore make it possible to reuse experts' design rather than reinventing the wheel, they fail to address one important application development aspect: code reuse. Design patterns must be implemented repeatedly, which can be not only time-consuming but also error-prone. For

example, certain design patterns — for instance, the Reactor pattern [35] — involve complex structures and interaction behaviours among different participating components. To overcome such a problem, a system called CO_2P_3S[1] has been developed at the University of Alberta. CO_2P_3S takes a generative application-development approach based on design pattern templates [4]. Each template covers one set of closely related problems and provides a code-base for application generation. For example, the Mesh pattern template in CO_2P_3S addresses the set of problems related to parallel matrix computations. The generated applications are completely in Java, and therefore are portable across different hardware and operating system platforms. Further, by setting template options, a pattern template can be customized to one specific problem in the problem domain. The customized template is used to generate an object-oriented framework. After weaving hook methods into the generated framework through the built-in tools support offered by CO_2P_3S, complete applications can be created.

Prior to this dissertation, work in CO_2P_3S has been conducted in the domain of parallel programming, resulting in a collection of basic parallel programming templates that are capable of generating parallel programs for all the problems contained in the Cowichan problem set [5]. CO_2P_3S greatly eases the task of parallel program development. To create a parallel program using CO_2P_3S, users only need to select a parallel design pattern template and provide a few hook methods that contain the problem-specific sequential code. This means that users don't have to worry about the tricky parts in most of the parallel programs such as process synchronization, load balancing, and termination conditions. The parallel programs generated by CO_2P_3S not only have very high ratios of generated code vs. programmer-supplied code, but also achieve comparable performance against their handcrafted counterparts.

The success of CO_2P_3S in the parallel programming domain has greatly motivated the research described in this dissertation. This research applies CO_2P_3S to the network server application domain. The goal of this research is to create a design pattern template for the construction of network server applications and evaluate the effectiveness of the

[1] Correct Object-Oriented Pattern-Based Parallel Programming System, pronounced as "cops".

template-based code generation approach of CO_2P_3S. The resulting design pattern template must be able to greatly ease the complexities involved in the construction of network server applications similar to those basic parallel design pattern templates — its direct predecessors — have achieved in the parallel programming domain: minimizing the coding efforts while achieving good application performance.

1.2 The scope of this thesis

To achieve the goal outlined in the previous section, a design pattern template called Network Server (N-Server) was constructed. It employs an event-driven approach as its concurrency model and uses the Reactor design pattern as the fundamental mechanism for event demultiplexing and dispatching. I/O operations must be nonblocking for event-driven concurrency. To meet such a requirement, Java NIO [6] is employed for nonblocking socket I/O, and nonblocking file I/O operations are simulated using a pool of threads (because they are not yet available in the Java API). Many design options are offered, some of which exist purely to address performance issues while others deal with features demanded by many network server applications. They include event logging, performance profiling, event scheduling, file caching, and overload control. The existence of these options makes the N-Server highly customizable and suitable for the construction of a large variety of network server applications, ranging from trivial cases like a Time server to those as sophisticated and performance-sensitive as an HTTP[2] server. Using the N-Server to create a network server application is fairly easy, because the user only has to supply a few event handlers.

To evaluate the N-Server, two server applications, an HTTP server called COPS-HTTP and an FTP[3] server called COPS-FTP were constructed. Only a small fraction of code needs to be programmed for these two servers, while the rest of code is either generated from the N-Server or reused from existing application libraries. Experiments have been done to compare the performance of COPS-HTTP with Apache Web server, a present

[2] Hypertext Transfer Protocol
[3] File Transfer Protocol

state-of-art Web server. It was found that comparable levels of throughput and responsiveness are achieved.

In addition to the success of this research in demonstrating the effectiveness of applying the CO_2P_3S code generation approach in the network server application domain, a few areas of potential future work have also been proposed.

In summary, the major contributions of this dissertation research include the following:

1. The construction of the N-Server, a design pattern template that can be configured to automatically generate a large number of network server applications varying greatly in their functionalities and performance requirements.

2. The automatic generation of an HTTP server and an FTP server using CO_2P_3S. These are the first non-trivial applications generated by CO_2P_3S. They are also the first HTTP server and FTP server created through an automatic code-generation approach rather than handcrafting.

3. The re-evaluation of the pattern-based code-generation approach of CO_2P_3S and the provision of case-study experiences and suggestions for the future enhancement of CO_2P_3S.

1.3 Dissertation organization

This dissertation is composed of eight chapters. Chapter 2 introduces CO_2P_3S, describing its layered design philosophy, major features and components. It also describes MetaCO_2P_3S, which is used for the construction of design pattern templates, and it describes the generative application development process of CO_2P_3S. Chapter 3 is dedicated to concurrency issues. This chapter first compares several widely used concurrency models: dedicated thread/process, pre-allocated pools of threads/processes,

and event-driven concurrency. Then, it surveys several related systems: SEDA [10], SPED [11] and MPED [12]. Pioneering work in concurrent and communication design patterns has greatly influenced this research. Hence, Chapter 4 introduces several important design patterns in the domain of network server applications, including the Reactor, the Proactor [7], the Connector-Acceptor [8], and the Asynchronous Completion Tokens [9]. Chapter 5 documents the N-Server, including its analysis, design, features and options. It also describes how to use Server to generate a network server application. Chapter 6 describes two case studies using the N-Server, an HTTP Server and an FTP Server. Chapter 7 presents many performance experiments based on industry benchmarks like SpecWeb99 [13], designed to compare the performance of generated network server applications against their handcrafted counterparts. This chapter also describes artificial experiments designed to demonstrate additional features provided by N-Server, such as event scheduling and overload control. Finally, Chapter 8 presents the conclusion of this dissertation and gives a direction to future work.

Chapter 2

CO_2P_3S Overview

2.1 Introduction

CO_2P_3S is a software development system developed at the University of Alberta. It uses a generative software development approach based on design pattern templates.

Traditional design patterns do not have any implementation support, which makes it hard to use them for the software application development. There are many proposals intended to overcome this shortcoming [14, 15, 16, 17], and CO_2P_3S uses a generative approach that is based on design pattern templates. A design pattern template [18] contains a code base that can be instantiated to different implementations of a design pattern. It also records the descriptive information regarding implementation variations of a design pattern in the form of template options.

In CO_2P_3S, the generated design pattern instances are completely in Java. Java was chosen because it has the following properties: platform-independence, type safety, built-in support for concurrency, and code distribution.

CO_2P_3S is able to generate both sequential and parallel programs. For sequential programs, several design patterns (such as the Observer, the Composite, and the Decorator design patterns [2]) are currently supported, and others can be implemented easily.

Parallel programming can significantly speed up and scale up computationally intensive tasks. However, writing parallel programs can be a difficult task for a programmer, either because they lack the required knowledge and experience, or because parallel programming is more complex than sequential programming. Parallel design pattern

templates [4]are included in CO_2P_3S to assist the development of parallel applications. The generated parallel programs in CO_2P_3S can be executed either on a shared-memory multiprocessor computer or a distributed-memory environment such as on a network of workstations [20].

Section 2.2 gives some background on the three key technologies that are used in CO_2P_3S to improve the productivity and quality of software application development. The generative application development approach of CO_2P_3S is based on design pattern templates, which are described in Section 2.3. Section 2.4 describes the application development process and the layered application design philosophy of CO_2P_3S. Section 2.5 describes MetaCO_2P_3S, a tool used to facilitate the development and maintenance of design pattern templates.

2.2 Background

Three closely related techniques are widely used in the software industry to improve the productivity and quality of the sequential application development. They are object-oriented programming languages, object-oriented frameworks and design patterns.

Object-oriented programming languages use objects as the basic building blocks, with each object representing a real-world entity. This allows the application design and analysis to share the same notions, and therefore, minimizes the gap between them. Object-oriented programming languages support data encapsulation, polymorphism, and inheritance. Encapsulation brings about several desirable properties of software application design, such as data independence, abstraction, and separation of concerns. Polymorphism eases the complexities of software development by allowing objects from different classes to be used in the same context. Inheritance supports design reuse and code reuse by allowing subclasses to inherit interfaces and implementations from their super classes. Objects and classes can be grouped into class libraries for reuse. Although

[4] Parallel design patterns are expert solutions to recurring parallel program design problems. Just as a design pattern template instantiates a design pattern, a parallel design pattern template instantiates a parallel design pattern.

class libraries can speed up software development, they fail to preserve the information regarding the interaction among objects and the control flow of software applications. In other words, class libraries do not support the reuse of software application architectures — another valuable asset of software design and development. To overcome such a shortcoming, object-oriented frameworks and design patterns are often used.

An object-oriented framework[5] contains a collection of collaborating objects that provide the common functionalities for a family of software products. To develop a new software application with a framework, users only have to specialize it by providing application-specific features. Often, this is done by providing hook methods or subclassing existing classes in the framework. A framework can greatly speed up software design and development, because it supports the reuse of both the application design and the implementation. A good framework has very stringent requirements. A framework must be as flexible and general as possible, because it is not designed for one single product but a family of products. Being not only a valuable asset but also the result of long-term development efforts and investments, a framework is likely to be used for a long period of time. Therefore, it must be initially designed for changes and able to evolve over time. To build a good framework is a difficult task and the adoption of design patterns helps to improve the productivity and quality of framework design and development.

Design patterns, are generic solutions to recurring software design problems. Like frameworks, design patterns also promote design reuse at the architecture level. However, design patterns deal with smaller architectures than frameworks. A framework may contain several design patterns, while a design pattern describes how to structure the subsystems and components of a framework to make a good framework design. Design patterns capture experts' design experiences and help to document and communicate those experiences. With design patterns, junior developers can learn from their seniors' best practices, and therefore, they can greatly improve their application design and development. Design patterns have one severe limitation: unlike frameworks, they do not

[5] For brevity, the word "framework" will be used later on in place of "object-oriented framework", whenever there is no ambiguity.

support code reuse. Design patterns often exist in the form of documentation. Whenever a design pattern is put to use, it has to be implemented again. For fairly simple design patterns, it is straightforward but tedious to implement them. For design patterns that involve complex structure and/or interaction behaviours, their implementation can be not only time-consuming but also error-prone.

2.3 Design Pattern Templates

Traditional design patterns provide documentation support but no implementation support, which is mainly caused by the fact that a design pattern is a generic solution to a family of design problems. Although each problem in this family shares a basic structure, a design pattern must be adapted to its specific context. Adapting a design pattern may require structure variations or the addition of problem-specific behaviours. Sometimes, a design pattern may even derive its interface from the specific application context. These adaptation requirements of design patterns have basically excluded the possibility of using a static object-oriented framework to realize a design pattern. When a static framework can be used, it often must rely on indirection code between some objects to achieve the structural variations of a design pattern, and this adds performance overheads.

In view of the implementation requirements of design patterns, CO_2P_3S uses a generative approach, whose basis is a design pattern template. A design pattern template can be used to instantiate a design pattern. It contains a code base that realizes the basic structure of a design pattern. It also provides a set of template options that can be chosen by the user to adapt the code base to a specific application context of the design pattern. These template options also serve as switches in the code base of the design pattern template to govern which piece(s) of code to generate. The code base of a design pattern template and the code of an instantiated design pattern have such a relationship: if the latter can be viewed as a static object-oriented framework, then the former is a dynamic framework whose structure is only fixed after it has been adapted to a specific application context of the design pattern.

Options in a design pattern template can be grouped into three categories based on their effects: lexical options, design options and performance options. A lexical option allows a user to specify names for participants in a design pattern. This avoids name clashes when a design pattern is used multiple times in an applications. A design option makes structural changes to the design pattern that a user must be aware of, while a performance parameter makes structural changes that are transparent to the user.

2.4 Development Process and Design Philosophy

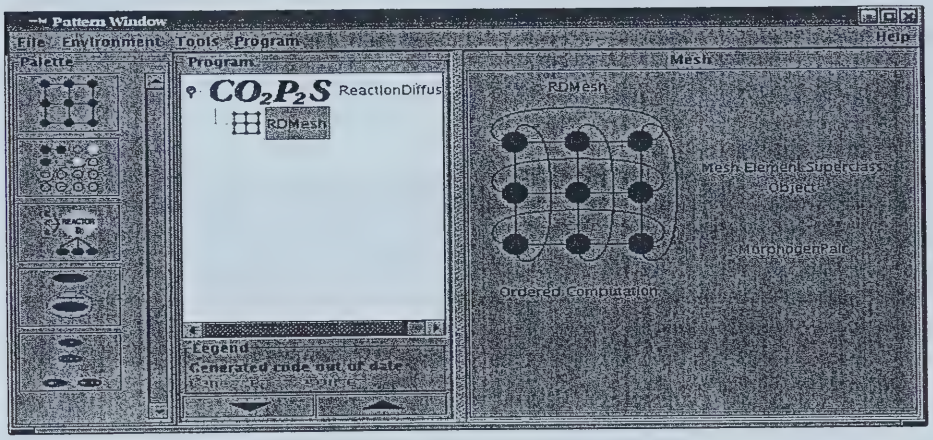

Figure 2.1: Selecting design pattern templates

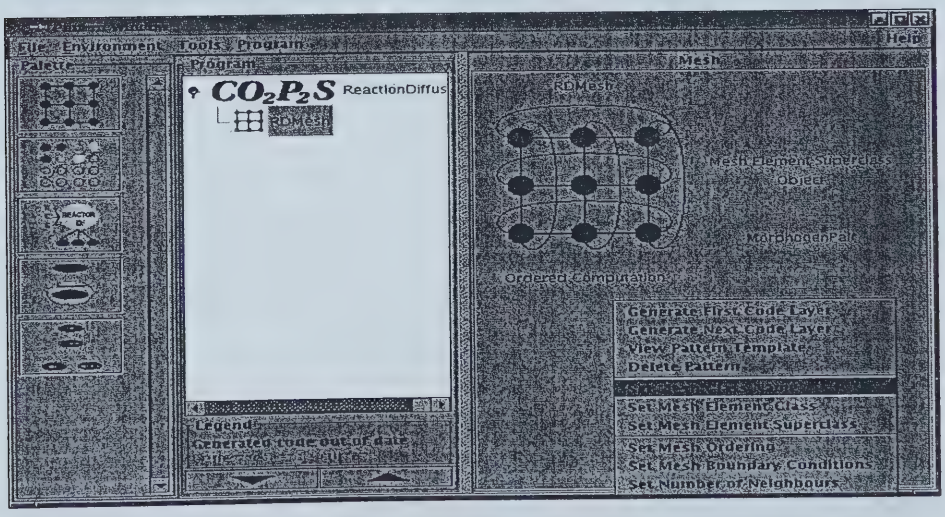

Figure 2.2: Setting options

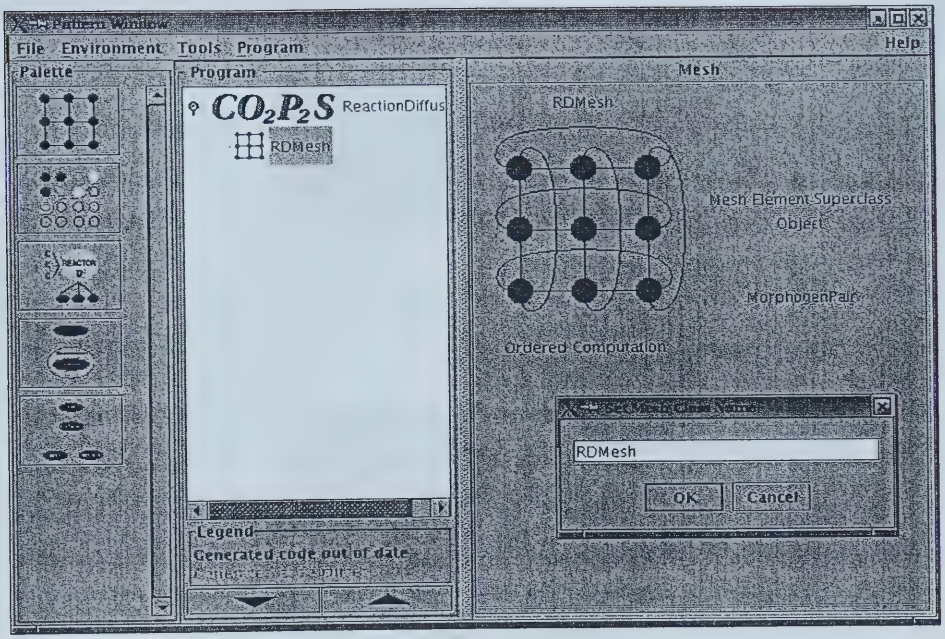

Figure 2.3: Setting a lexical option

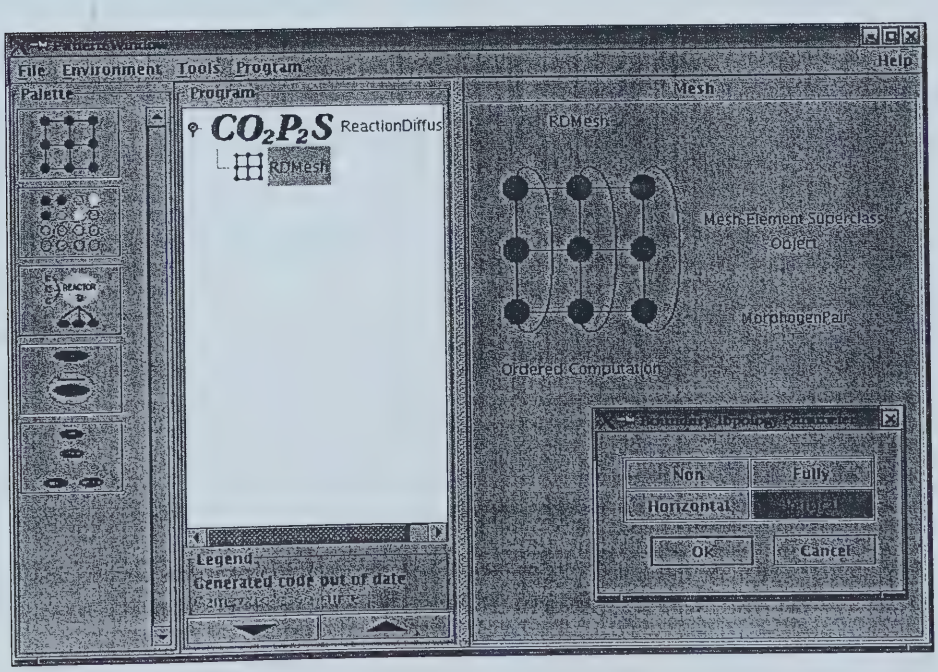

Figure 2.4: Setting a Design Option

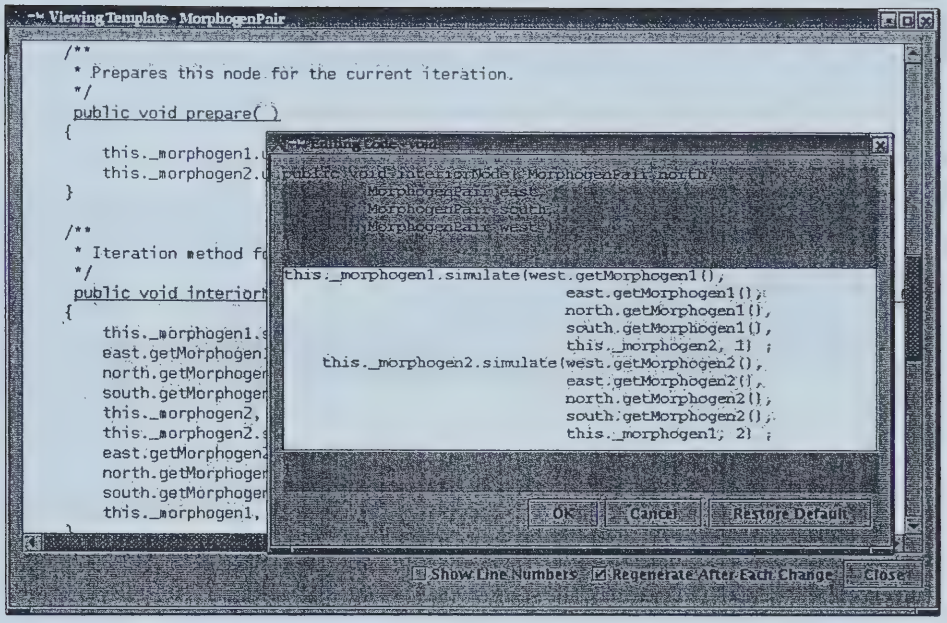

Figure 2.5: Insert hook methods

The generative application development process of CO_2P_3S has three steps, with each step facilitated by the IDE (Integrated Development Environment) of CO_2P_3S. The three steps are:

1. **Identify the design patterns.** Figure 2.1 shows the development of a parallel program using a parallel design pattern template called Mesh[6]. On the left side of the window, there are five design pattern templates, and users can choose any of them for their application. In this application, Mesh is chosen.

2. **Customize each pattern template by setting option values and generate an object-oriented framework that constitutes the skeleton of the application.** Figure 2.2 shows the list of template options provided by Mesh. Figure 2.3 shows the dialog for setting a lexical option. Figure 2.4 shows a dialog for a design option.

[6] Mesh is one of the parallel design pattern templates provided by CO_2P_3S.

3. **Insert hook methods to specialize the generated object-oriented framework and implement additional code required by the application.** Figure 2.5 shows one window used to edit a hook method.

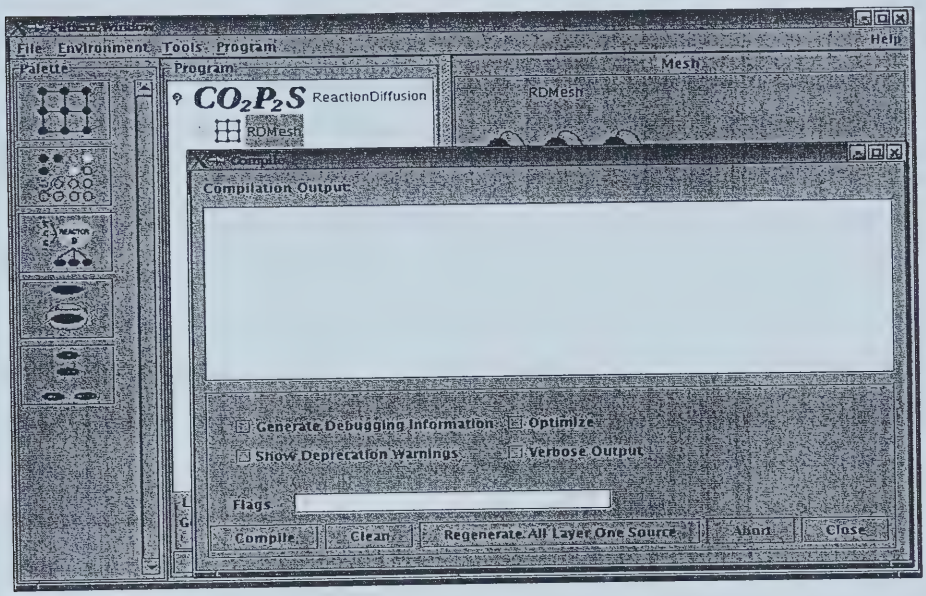

Figure 2.6: Compilation

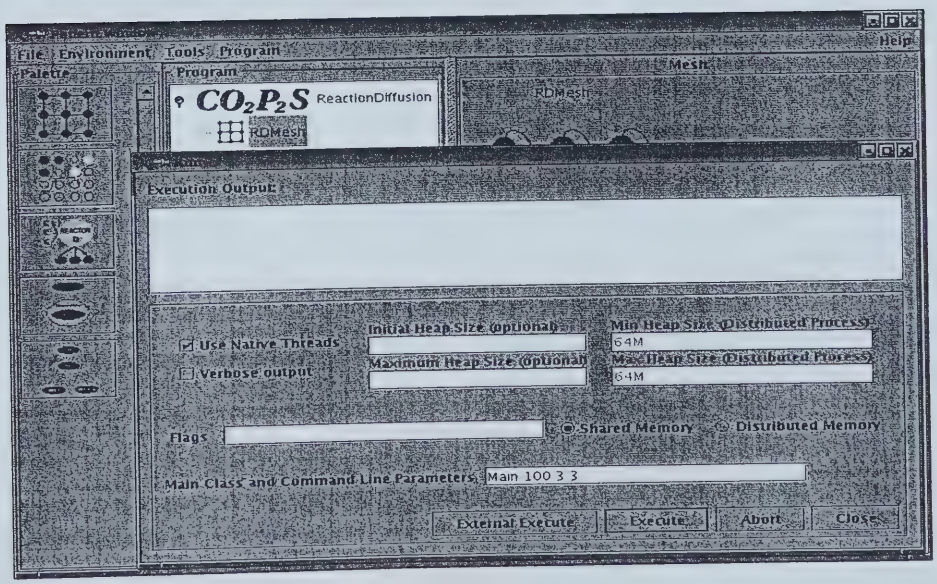

Figure 2.7: Execution

An application is completed after all of the above three steps have been done. CO_2P_3S also provides facilities for the compilation and execution of the generated application (shown in Figure 2.6 and Figure 2.7).

CO_2P_3S takes a layered application design philosophy, which is motivated by the observation that a design pattern template cannot cover all the variations of a design pattern, no matter how powerful it is. One reason is that a design pattern is generic and is very likely to have variations that are not anticipated by the template designer. Another reason is that there exists a trade-off between the usability and the flexibility of the design pattern template. From a user's perspective, a design pattern template that covers more variations is more flexible but harder to use than the one that covers fewer, because it may be more difficult to adapt it for use. This raises the issue of openness. Openness requires that the user should have access to the generated code that realizes a design pattern, so that the user can implement unsupported structural variations. Further, users may also need access to the whole generated code in order to tune the application to improve performance. In CO_2P_3S, users usually work at the pattern layer, which gives users the freedom to select design pattern templates, set template parameters, insert hook methods, and add application-specific code. However, the generated framework is completely open. This open code constitutes the code layer. Whenever needed, users can make any necessary modifications at the code layer to change the structure or improve the performance.

2.5 MetaCO_2P_3S

Although CO_2P_3S has provided a number of design pattern templates, it is far from being complete. On one hand, there are always applications that cannot be implemented by one fixed set of design pattern templates. On the other hand, new design patterns keep on emerging. Actually, this is a severe limitation that all the pattern-based programming systems are facing today. Facilities must be provided to allow users to add new design pattern templates and enhance existing design pattern templates.

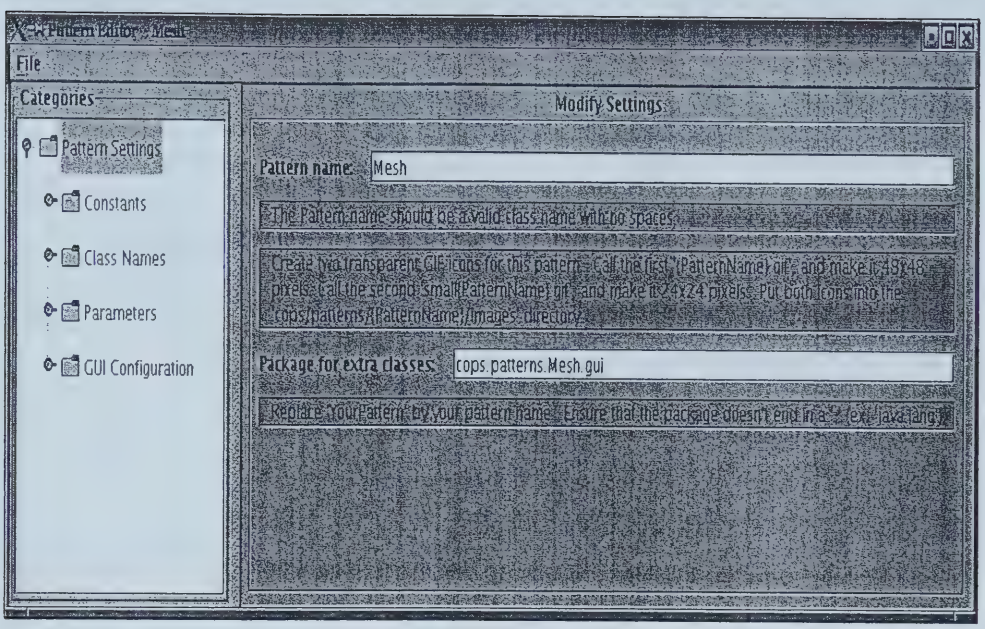

Figure 2.8: The Meta CO_2P_3S GUI

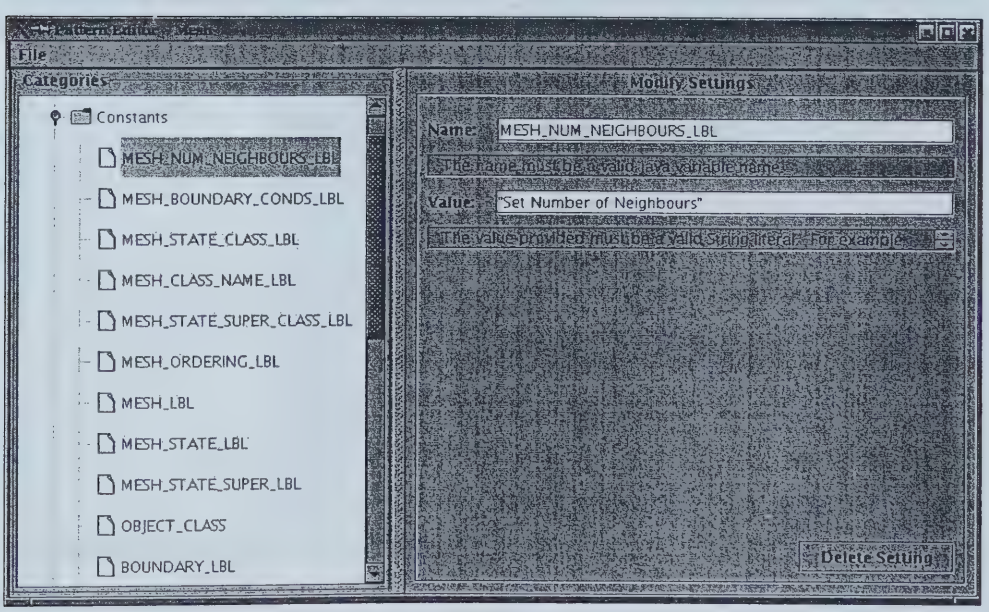

Figure 2.9: Template constants

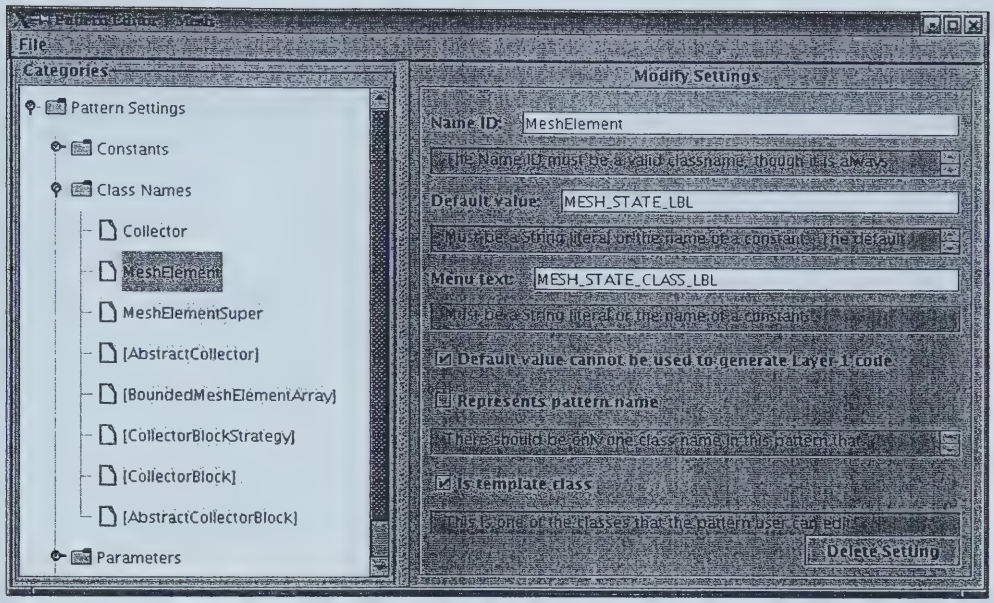

Figure 2.10: Template classes

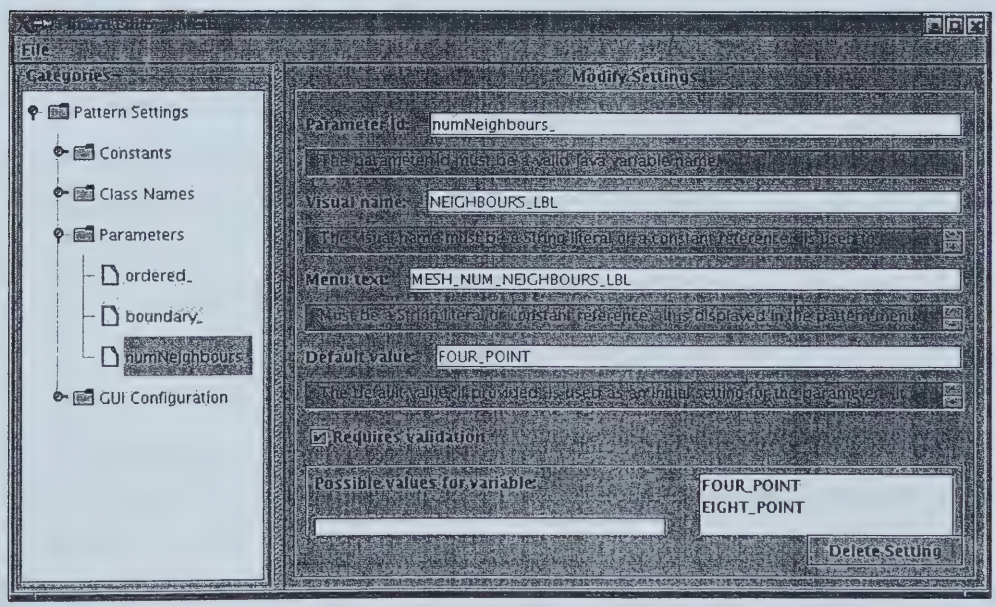

Figure 2.11: Template options

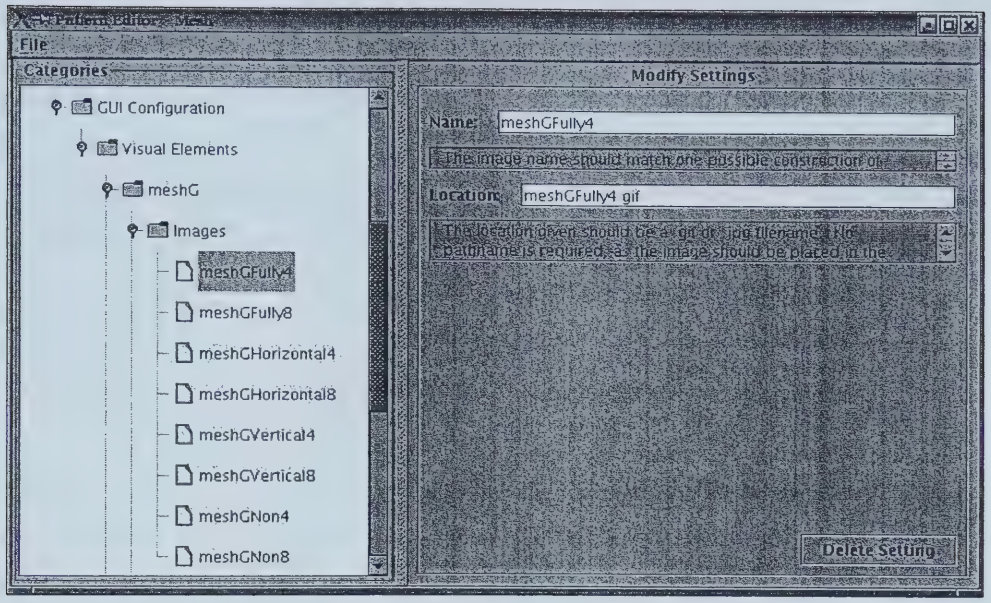

Figure 2.12: Template GUI configurations

MetaCO$_2$P$_3$S is a tool created to facilitate the development and maintenance of design pattern templates. In MetaCO$_2$P$_3$S, the content of a design pattern template is grouped into four sections: Templates Constants, Template Classes, Template Options, and Template GUI[7] Configurations (shown in Figure 2.8). The Template Constants section is a repository that stores all constants used by the design pattern template. Figure 2.9 shows the window used for the constant definition. The Template Classes section contains a collection of classes that constitute the code base of the design pattern template (shown in Figure 2.10). The Template Parameter section is used to facilitate the definition of design pattern template options. Figure 2.11 shows the window used for the definition of a basic option. The Template GUI Configurations section helps the template designer to specify the presentation of the design pattern template in CO$_2$P$_3$S (shown in Figure 2.12).

MetaCO$_2$P$_3$S stores a design pattern template as an XML[8] document to facilitate the sharing and exchange of design pattern templates. A DTD[9] has been created to verify the

[7]GUI is the acronym of Graphical User Interface.
[8]XML stands for eXtensible Markup Language. It's an industry standard approach for the sharing and exchange of structured or semi-structured documents.

format of CO_2P_3S design pattern templates. Whenever a design pattern template is imported into CO_2P_3S, an XSL[10] document will be used to convert it from an XML document to Java source files, which can be compiled and loaded into CO_2P_3S.

To sum it up, MetaCO_2P_3S can greatly ease the complexities involved in the development of design pattern templates and is an integral part of CO_2P_3S.

[9] DTD is the acronym of Document Type Definition.
[10] XSL stands for eXtensible Stylesheet Language.

Chapter 3

Concurrency

3.1 Concurrency Models for Server

A server application is concurrent if it has the capacity to handle multiple requests simultaneously. Based on how concurrency is achieved, concurrent server applications can be grouped into two broad categories: multiprogramming and event-driven.

Multiprogramming concurrency models are based on either multiprocessing or multithreading. Multiprocessing uses multiple processes in an operating system to achieve concurrency. A process manages the program instruction execution context and a collection of resources, such as virtual memory, I/O, and signal handlers. Each process has its own address space, which prevents processes from interfering with each other. However, this also leads to a high memory demand. Multithreading alleviates this problem by allowing multiple threads to share the same process address space. Each thread has its own instruction pointer and local stack. The creation and switching overheads of threads are much lower than those of processes.

In multiprogramming concurrency models, whenever the execution of a process or a thread is paused due to blocking operations, such as disk I/O, the CPU can switch to another process or thread, so that multiple requests are served simultaneously. Multiprogramming can easily take advantage of the presence of multiple processors.

The two most commonly used multiprogramming concurrency models are the process-per-connection model and the thread-per-connection model. The former is based on multiprocessing, while the latter is based on multithreading. In these two models, a process/thread is dedicated to serving requests from a client connection. This process/thread can be created on the fly or allocated from a pool of worker processes/threads. Using a pool of worker processes/threads is more popular, because it reduces the response time by avoiding the process/thread startup overheads [42].

Although multiprogramming concurrency models are relatively straightforward to implement, they don't scale up very well. When the number of processes/threads is large, the overheads associated with multiprocessing/multithreading — including process/thread scheduling, cache misses, and lock contention — can cause serious performance degradation. When the number of processes/threads is bounded, additional connections are not accepted immediately, so a backlog is created. This leads to not only high response latency but also unfairness to clients [10]. In addition, it's hard to support Quality of Service using a multiprogramming concurrency model. The reason is that multiprogramming concurrency models have very limited capacity for resource management and request scheduling [25, 26]. The design of traditional OS schedulers focuses on supporting the concurrent execution of multiple processes/threads in a time-sliced manner. In these operating systems, although the CPU usage can be scheduled by setting the priority-level of processes/threads, there are no scheduling mechanisms for the usage of other important system resources, such as disk I/O and network bandwidth.

To overcome these limitations, event-driven concurrency models are becoming more popular. Event-driven concurrency models have the following characteristics:

- Application behaviour is triggered by internal or external events. Events arise from multiple event sources, such as device drivers, I/O ports, sensors, timers, and other application components. Because events that correspond to the steps needed for processing a request can be manipulated directly, event-driven applications have more control over resource management and request scheduling.
- There are a small number of threads that loop continuously to process events. Each thread is usually mapped directly to a CPU. This avoids the thread switching and scheduling overheads associated with multiprogramming concurrency models, and therefore, improves the application performance. To prevent threads from blocking, events must be asynchronous. However, some asynchronous event mechanisms like disk I/O are not supported by many modern operating systems. Unfortunately, this negates the performance and scalability advantage of event-

driven applications, unless more creative thread management is used (see section 3.2.3).

- The handling of each concurrent request is often modeled as a Finite State Machine (FSM), which controls the logic of event processing and detects illegal transitions.

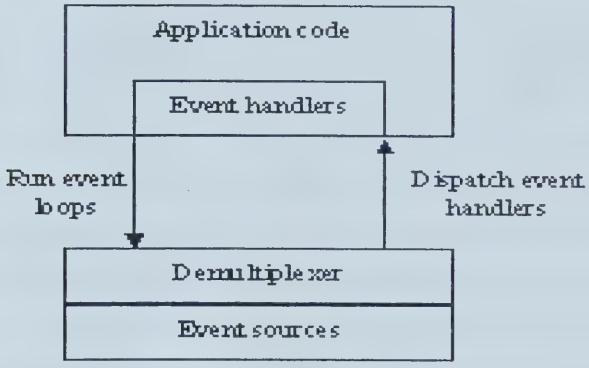

Figure 3.1: Event-driven application layers

Event-driven applications often have a layered application architecture (shown in Figure 3.1), which is composed of the following layers [27, 28]:

- Event sources. Detect and retrieve events from multiple hardware and software components.
- Event demultiplexer. Waits for events to arrive from multiple event sources and dispatches them to their event handlers.
- Event handlers and application code. Perform application-specific processing in response to events.

This layered application architecture separates the concerns of event demultiplexing and event handling. To build a concurrent application that uses an event-driven concurrency model, developers can focus on implementing the event handler and application code layer, and reuse the event sources layer and the demultiplexer layer.

3.2 Examples of Event-Driven Concurrency Models

Event-driven concurrency models are becoming increasingly pervasive. This section describes three event-driven concurrency models: single-process-event-driven, multi-process-event-driven, and staged-event-driven-architecture.

3.2.1 Single-Process-Event-Driven

The single-process-event-driven (SPED) model uses a single process to handle multiple requests simultaneously. This process uses system calls like the BSD UNIX *select* or the System V *poll* to check for ready I/O events. These I/O events correspond to the basic steps of processing a request. For instance, the *socket readable* event signals a server that an event has arrived for processing, while the *socket writable* event signals that a server can send a reply to the remote peer. The I/O operations indicated by these I/O events are nonblocking, so that a server can handle multiple requests iteratively. This is how the SPED achieves concurrency. However, SPED does not take advantage of multiple processors. Known uses of the SPED model include the Zeus Web Server [11] and the Harvest/Squid hierarchical web cache [29].

The SPED model can achieve good performance by avoiding synchronization and process/thread switching overheads, because only a single process is used. However, it cannot be applied, if some I/O operations are blocking, since it would also suffer from severe performance degradation. Even when operating system claims to be nonblocking, blocking can still occur. For instance, nonblocking disk I/O operations may actually block on most versions of UNIX [12].

3.2.2 Multi-Process-Event-Driven

The multi-process-event-driven concurrency model (MPED) inherits the spirit of SPED by using a main process to handle concurrent events one after another, and overcomes its limitation by using multiple *helper* processes/threads to handle blocking I/O operations. When a blocking I/O operation is needed, the main process passes this operation to a

helper through an interprocess communication (IPC) mechanism, such as a pipe. Although the *helper* may block when performing this operation, the main process can continue to handle other requests. After the blocking I/O operation is completed, the *helper* notifies the main process by sending an event through IPC, and the main process learns of this event using the *select* or *poll* just like other I/O events. The Flash web server [12] makes use of the MPED model.

The MPED model suffers from the same problem as the SPED model. That is they don't take advantage of the presence of multiple processors in a machine. Therefore, they do not scale up very well, especially when they face CPU-intensive workloads.

3.2.3 Staged-Event-Driven-Architecture

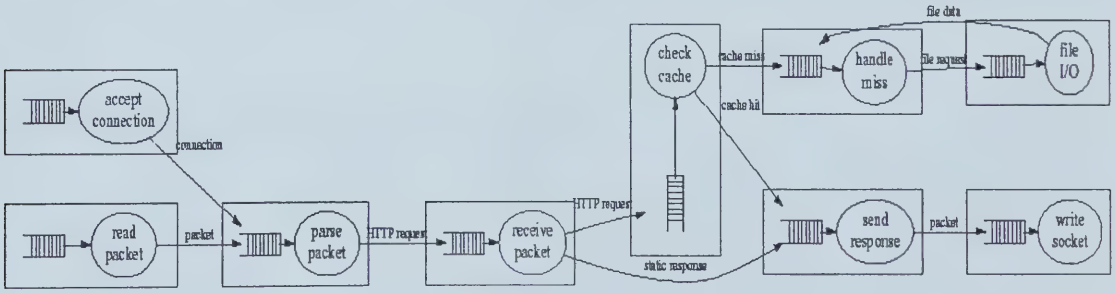

Figure 3.2: A staged-event-driven HTTP server

The staged-event-driven-architecture (SEDA) emphasizes the fact that a FSM is often used to model the handling of each concurrent request in event-driven concurrency models. In SEDA, an FSM stage is embodied as a self-contained application component, which consists of an event handler, an incoming event queue, and a pool of threads. These threads operate concurrently with each other. They pull events off the incoming event queue and dispatch them to the event handler. A controller can dynamically adjusts the thread pool size of each stage. When the number of events waiting in the event queue of a stage for processing exceeds a certain user-specified threshold, more concurrency is needed. In this case, more threads can be added. On the other hand, if a thread is idle for

a period of time, it can be removed to release system resources. A set of stages is also allowed to share the same pool of threads. The event handler encapsulates the specific application logic corresponding to the stage that handles a request. It processes each event and queues new events onto event queues of other stages. Figure 3.2 shows a SEDA-based HTTP server, in which the handling of an HTTP request is decomposed into different stages separated by queues. These stages include: accept-connection, read-packet, parse-packet, receive-packet, check-cache, handle-miss, file-I/O, send-response, and write socket. Similar to the MPED model, SEDA also use a pool of threads to simulate the existence of nonblocking I/O operations. Automatic overload control is a feature required by many high-performance concurrent server applications. The SEDA model addresses overload control by setting a rate that events can be queued onto the incoming event queue of each stage. If this rate is exceeded, additional events are discarded automatically. A prototypical implementation of the SEDA is called Sandstorm, which is used to implement the Haboob web server and the Gnutella packet router [10, 29].

The design of the SEDA model has the following problems:

- A SEDA-based application is very likely to be decomposed into more stages than processors. In such a case, the total number of threads in the application is greater than the number of processors, since each stage has an individual thread pool. This inevitably adds unnecessary thread switching overheads and therefore sacrifices the performance advantage of event-driven concurrency models. Although multiple stages can be configured to share the same thread pool, they still do not function the same as a stage. Consequently, features like dynamically adjusting the thread pool size and automatic overload control cannot be applied to these stages. On the other hand, there is also a significant amount of synchronization overhead, because accesses to each incoming event queue must be synchronized. If these stages are merged into one single stage, the application modeling and structuring advantage of the SEDA, derived from its close resemblance to a FSM, is no longer preserved.

- Events queued into a stage that belong to the same request can be processed out-of-order. This can cause tricky application errors that are quite difficult for the programmer to detect. Even though the programmer is aware of this problem, code still needs to be rewritten to handle it, which increases the development complexity of SEDA-based applications.
- The automatic overload control schema of the SEDA model allows individual stages to make an event-discarding decision independently. This can make application-layer communication protocols harder to design, since a client must deal with dropped events that can happen when a server is overloaded. This problem becomes more serious, when a request requires several exchanges of messages between the client and the server.

3.3 Summary

Event-driven concurrency models have many advantages over multiprogramming concurrency models, and therefore, they have become more pervasive. However, there still exist several challenges for the design and implementation of event-driven concurrency models. It is hard for an application developer to implement an event-driven concurrency model correctly. The lack of OS support for nonblocking I/O mechanisms negates the performance advantage of event-driven concurrency models. Therefore, a poorly designed event-driven concurrency model can fail to achieve good performance. In conclusion, employing event-driven concurrency models in a server may be an effective approach to achieving high performance, if more research solves the complexity and correctness issues. The research described later in this dissertation reduces the complexity and improves correctness of servers based on event-driven concurrency models by generating the complex code for the programmer.

Chapter 4

Network Server Application Design Patterns

A network server application is more complex to design and develop than a stand-alone application, because it must deal with many problems typical to networked and concurrent applications. The Network Server (N-Server) design pattern template, developed in the research described in this dissertation, reduces the complexity by generating most of the code needed by an event-driven network server application. Like other design pattern templates in CO_2P_3S, the N-Server is also based on design patterns. Four design patterns, dealing with how an event-driven network server application initiates, receives, demultiplexes, dispatches, and processes events, have greatly influenced the design of the N-Server. These four design patterns are the Reactor [35], the Proactor [7], the Acceptor-Connector [8], and the Asynchronous Completion Tokens [9]. This chapter introduces these four design patterns.

4.1 Reactor

A network server application must handle requests delivered from one or many clients. The Reactor design pattern documents a solution to the design problem of how an event-driven network server application demultiplexes and dispatches these events. It is composed of the following key participants:

- **Handles:** Identifies resources managed by the OS and encapsulates methods to access them. Common resources include sockets, timers, synchronization objects, etc.
- **Event Handler:** Specifies an interface consisting of a hook method that must be implemented to provide application-specific services.
- **Dispatcher:** Takes charge of registering, removing, and dispatching Event Handlers to process ready events. It delegates the polling of new events to the Synchronous Event Demultiplexer.
- **Synchronous Event Demultiplexer:** Polls a set of handles for events to occur. It blocks waiting until there are ready events.

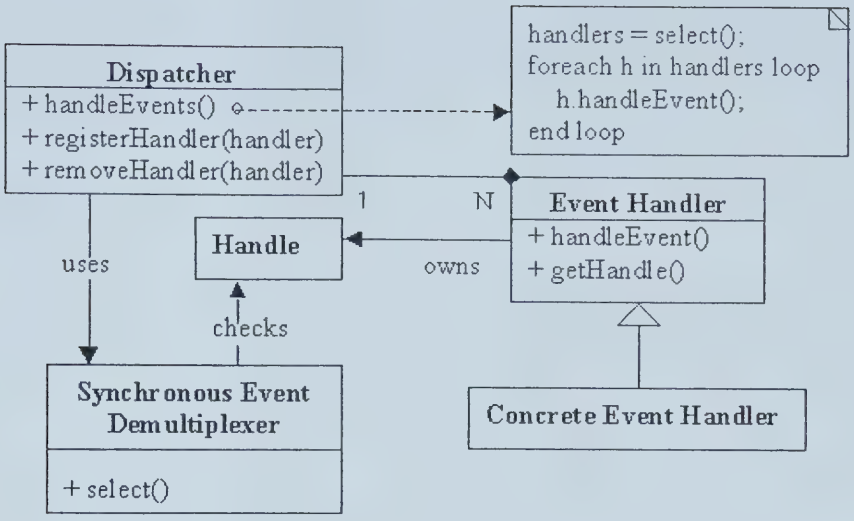

Figure 4.1: Reactor

Figure 4.1 illustrates the interaction of these participants. A number of Concrete Event Handlers are registered with the Dispatcher. Each Concrete Event Handler owns a Handle and implements the hook method *handleEvent*. The Dispatcher invokes the *select* method of the Synchronous Event Demultiplexer to wait an event to occur on any of the registered Handles. When an event occurs, the Synchronous Event Demultiplexer notifies the Dispatcher to call back to the *handleEvent* hook method of the Event Handler to perform application-specific functionalities in response to the event.

The Reactor design pattern offers several benefits, such as separation of concerns and improved reusability and modularity. It supports separation of concerns by decoupling the event demultiplexing and dispatching mechanism from the application-specific functionality. The event demultiplexing and dispatching mechanism is application-independent and can be implemented as a reusable component. The application-specific functionalities can be decomposed and encapsulated in several Event Handler classes. This improves the modularity of a network server application. The Reactor design pattern suffers from one severe limitation: it does not scale up very well, because it serializes all event handling at the event demultiplexing layer.

The N-Server employs the event demultiplexing and dispatching mechanism of the Reactor design pattern and overcomes its performance limitation by using a pool of threads to perform event processing (described later in Chapter 5).

4.2 Proactor

The Proactor design pattern provides another event demultiplexing and dispatching mechanism that is different from the Reactor design pattern. It does not passively wait for events to arrive and then react, like the Reactor. Instead, it associates a Completion Handler with an asynchronous operation. The Completion Handler encapsulates application-specific functionalities to handle a client request. Upon completion of the asynchronous operation, the associated Completion Handler is dispatched to process the result of the operation.

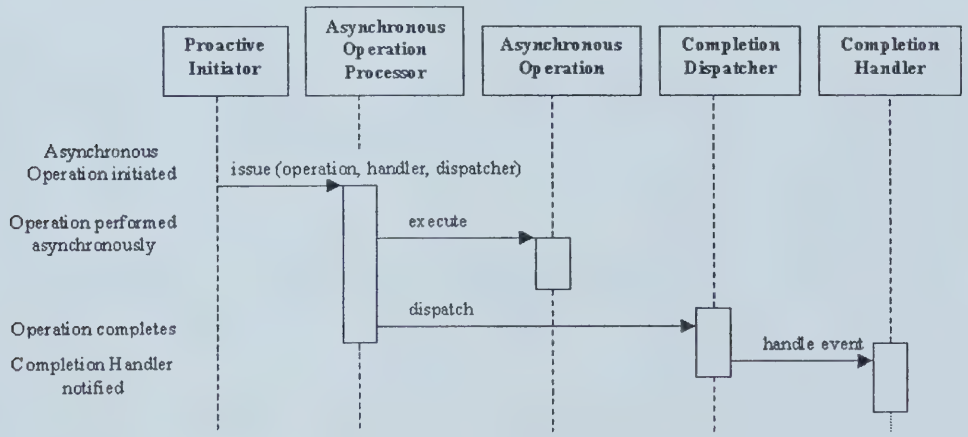

Figure 4.2: Interaction diagram of the Proactor design pattern

The key participants of the Proactor design pattern include the following:

- **Proactive Initiator:** Can be any entity in an application that initiates an asynchronous operation. It associates a Completion Handler and the Completion Dispatcher with the Asynchronous Operation.

- **Completion Handler:** Contains a hook method that encapsulates application-specific functionalities that are executed in response to the completion of an asynchronous operation.
- **Asynchronous Operation:** Identifies an operation (such as I/O and timer operations) issued by an application. The application thread is not blocked when executing an Asynchronous Operation.
- **Asynchronous Operation Processor:** Executes Asynchronous Operations on behalf of an application. This component is typically part of the OS. However, it can also be implemented using a pool of threads.
- **Completion Dispatcher:** Is responsible for calling back the Completion Handler when an Asynchronous Operation completes.

The interaction of these participants is shown in Figure 4.2.

The Proactor design pattern not only offers the same benefits as the Reactor design pattern, but also alleviates the performance limitation of the Reactor design pattern by allowing multiple Asynchronous Operations to execute concurrently. Unfortunately, the Proactor design pattern has a primary drawback. Most modern operating systems except Windows/NT do not directly support the dispatching of associated Completion Handlers upon the completion of Asynchronous Operations, although almost all of them support some form of asynchronous I/O. Thus, a programmer must implement the Completion Dispatcher explicitly. One possible implementation is to use the Reactor design pattern, which polls the completion of Asynchronous Operations using the Synchronous Event Demultiplexer and dispatches the associated Completion Handler. Such a mechanism is called Post-reactive dispatching. The N-Server uses this implementation to simulate nonblocking disk I/O operations, which are not available in the current release of JDK.[11]

4.3 Acceptor-Connector

Connection establishment and data communication are two independent aspects of a network application. Data communication can be carried out regardless of how connections are established.

[11] JDK is the acronym for Java Development Kits.

Connection establishment can be affected by issues like: (1) which one initiates the connection and (2) which network protocol is used. The two network peers play asymmetrical roles when establishing a connection: one sends a connection request actively, while the other waits for a connection passively. Once the connection is established, communication can occur regardless of which peer initiated the connection. Also, the network protocol becomes irrelevant once connections are established. This is because different network programming interfaces (such as sockets and TLI[12]) use uniform operations (e.g. send/recv) to exchange data between network peers, although they may differ greatly when establishing connections.

Decoupling these two aspects allows choices about them to be made independently. This improves the application modularity and reduces the development and the maintenance cost of network applications.

The Acceptor-Connector design pattern decouples connection establishment in a network application from data communication. It is often used in conjunction with the Reactor or the Proactor. When layered on top of the Reactor design pattern (as the N-Server), it has the following participants:

- **Service Handler:** Contains a hook method that is called by the Acceptor or Connector to initialize a service when a connection is established. A typical service initialization registers Event Handlers to process events arising from the established connection.
- **Acceptor:** Is an Event Handler that is used to establish a connection passively. When a new connection request arrives, it is dispatched to accept the new connection and call the Service Handler to initialize the service.
- **Connector:** Is an Event Handler that is used to establish a connection actively. When an active connection operation is completed, it is dispatched to call the Service Handler to initialize the service.

[12] Transport Level Interface.

4.4 Asynchronous Completion Token

An event-driven application often issues asynchronous operations to acquire one or more services (e.g. disk I/O), and a service indicates its completion by sending back a response. Appropriate actions must be performed to process the response. Thus, there should be a demultiplexing mechanism that associates service responses with the actions to be performed. When a service response arrives, this mechanism not only needs to quickly determine the action(s) and state needed to process the response, but also must recover the context in which the asynchronous operation was issued.

The Asynchronous Completion Token design pattern documents an effective mechanism to associate service responses with actions to be performed. This mechanism associates an asynchronous completion token (ACT) with an asynchronous operation. An ACT uniquely identifies the actions and state needed to process the operation's completion. When a service completes the operation, it notifies the completion by sending back a response together with the ACT sent originally, so that the state and actions needed to process the response can be easily identified.

The Asynchronous Completion Token can be used together with the Reactor or the Proactor. When used together with the Reactor, a service response may be modeled as an ACT Event that contains both the response and the ACT. An Event Handler that encapsulates the actions needed to process the response is registered with the Dispatcher (in this case, the ACT no longer needs to specify the actions). When an ACT Event arrives, the Event Handler is dispatched, and it can easily determine the response processing state and recover the context in which the asynchronous operation is issued. The N-Server uses the Asynchronous Completion Token together with the Reactor to simulate any nonblocking operations, such as disk I/O.

Chapter 5

The Network Server Pattern

5.1 Intent

The Network Server design pattern template (N-Server) is designed to generate network server applications. It is supposed to significantly ease the complexity of the network server application development, while satisfying a diverse range of performance and functionality requirements.

5.2 Motivation

To achieve high-performance and have a good control over resource management and request scheduling to support different levels of quality of service (QoS), the N-Server employs an event-driven concurrency model. The Reactor design pattern is used, because it describes an effective solution to the design problem of how a network server application demultiplexes and dispatches events delivered from one or many clients. Thus, the basic structure of the N-Server is based on the Reactor design pattern. However, the N-Server is not equivalent to the Reactor design pattern. In one way, it specializes the Reactor design pattern and in another way, it extends it. It specializes the Reactor design pattern by limiting it to a network communication server. It extends the Reactor design pattern by providing support to multiple event sources and multiple processors. Two participants are added to the Reactor design pattern, which are:

1. **The Event Source** Although the Reactor design pattern specifies how events should be demultiplexed and dispatched, it fails to address one important area: that is how to manage the event sources where events arise. In an application that uses the Reactor design pattern, events may arise from multiple event sources. For example, a network server application may have several event sources, such as a network interface, a timer, or some user-defined event source. Different event sources have different characteristics, and therefore, they should be managed

separately. Because it's not possible to anticipate and include all the event sources, there should be an effective mechanism to add new event sources. In N-Server, the solution to this problem is to add a new participant called the Event Source to the Reactor design pattern. The Event Source takes over the duties of managing different sources of events and polling their readiness. These duties belong to the Event Dispatcher in the Reactor design pattern. With the Event Source, event sources are managed separately and new event sources can be easily added.

- **The Event Processor** In the Reactor design pattern, all events are processed by the Event Dispatcher thread. This prevents network server applications based on the Reactor design pattern from taking advantage of the presence of multiple processors. A new participant, called the Event Processor, is added to solve this problem. An Event Processor contains a pool of threads that collaboratively carry out the task of handling ready events. In this case, the Event Dispatcher is only responsible for querying the Event Source for ready events and then passing those ready events to the Event Processor for processing. One limitation of event-driven concurrency models is that events must be nonblocking. However, there is often no nonblocking mechanism in the OS for many types of events, such as File I/O, database access, and synchronization. In particular, there is no nonblocking File I/O in Java, although nonblocking socket I/O is supported in Java in JDK1.4 and higher. This problem can be alleviated using the Event Processor, which uses a pool of threads to emulate the existence of nonblocking operations. For instance, if a user issues a File Read request event, one thread in the Event Processor will be allocated to handle the event. In this case, only that thread is blocked, and the remaining threads can still handle other blocking events. An Event Processor with one single thread can also be used to serialize accesses to an object whose access requires synchronization.

Although the addition of these two participants can greatly enhance the capability of the Reactor pattern, using such an extended Reactor design pattern template to construct a network server application is still quite inefficient. Users must explicitly write code to

deal with network communication, which is a difficult programming task. In addition, there are a large number of places where user code has to be supplied, such as the frequent registration and deregistration of different Event Handlers and the queuing of user-defined Events.

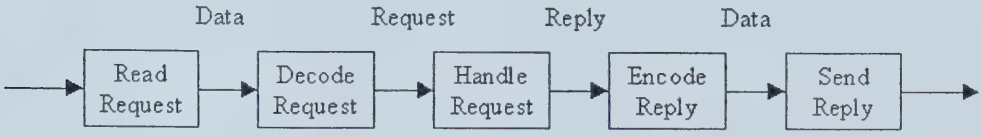

Figure 5.1: Five basic steps

This inefficiency is mainly caused by the inadequacy of the Reactor design pattern to exploit the similarities between network server applications. Most network server applications are similar in the way they establish network communication with peers and in the way they handle requests. To establish a network communication, they create a server socket listening to a certain network port for new connections. After a connection arrives, they accept it and iteratively carry out a five-step process to handle requests. These five steps are: Read Request, Decode Request, Handle Request, Encode Reply, and Send Reply (shown in Figure 5.1). The Read Request step reads the raw data of a request sent from the remote peer over the socket connection. The Decode Request step parses the request. The Handle Request step provides services by handling the request. The Encode Reply step encodes the result of the request in a form understood by the remote peer. The Send Reply step sends out the raw data of the result to the remote peer. Among the five steps, the Read Request and the Send Reply are almost the same across different network server applications, while the other three steps are application-dependent.

By generating code for establishing network communication and the two of the request processing steps, Read Request and Send Reply, the N-Server pattern can further reduce the complexity of network server application development. To develop a network server application with the N-Server pattern, the user only has to provide a few pieces of code corresponding to the non-common three processing steps: Decode Request, Handle Request, and Encode Reply. The N-Server generates code for the rest of the common

functionalities. This reduces coding effort and improves the correctness of the development of network server applications.

There exists a tradeoff between generality and efficiency in the N-Server. Without the inclusion of the network server application specific code (Read Request, Send Reply, and establishing a connection), the N-Server would be a template that instantiates the Reactor design pattern. Thus, it would be more generic and could be used for many types of applications, such as event-driven simulations and graphical user interface frameworks. With the inclusion of the network specific code, the N-Server becomes more specialized and more efficient for the automatic generation of network server applications. The N-Server sacrifices generality in favour of improving efficiency. This choice is inevitable, because it is mandated by the design goal of the N-Server.

Other functionalities that are often needed by many network server applications are also provided by the N-Server, including:

- **File Caching**: Many network server applications need to access disk files to provide services. However, disk I/O is slower by several orders of magnitude than memory access. To boost application performance, network servers often implement a file cache that keeps the disk file in memory for fast access. To relieve users from the burden of implementing a file cache, the N-Server can be configured to generate code that automatically caches disk files in memory. The caching capability provided by the N-Server is transparent to the programmer who uses the N-Server. This means that programmers have no extra development effort, unless specialized cache replacement policies are designed. Five cache replacement policies are provided by the N-Server:
    - **LRU**: The Least-Recently-Used cache entry is removed first.
    - **LFU**: The Least-Frequently-Used cache entry is removed first.
    - **LRU-MIN** [21]: To free space of size *S*, cache entries greater than *S* are removed first based on the LRU order; accordingly, in the order of *S/2, S/4, ...* .

- **LRU-Threshold** [21]: It is basically the same as LRU, except that objects larger than a certain threshold are never cached.
- **Hyper-G** [22]: Removes caches in the LFU order; ties are broken first by LRU, then by the size of the file.

Cache replacement policies that are different from these five can also be implement easily by the programmer.

- **Event Scheduling**: Many network server applications can benefit from the capability of altering the order in which requests from different concurrent connections are serviced. At least two kinds of benefits can be gained by this capability. They are: (1) improving the quality of service and (2) providing differentiated levels of services. In an event-driven network server application, such a capability requires the support of Event Scheduling. The N-Server can be configured to generate the necessary code to support an efficient Event Scheduling mechanism. In such a mechanism, the priority can be specified in whatever way the user wants, and events of higher priority are generally processed first, but events of lower priority are not starved.
- **Overload Control**: Overload in a network server application can cause increased response times, decreased throughput, and even service rejection. Event-driven concurrent server applications are extremely vulnerable to overload conditions, because they tend to accept as many connections as possible, which is not the case in server applications based on the process-per-connection or thread-per-connection concurrency models [23]. The N-Server can provide two mechanisms to control overload. The first one is trivial: that is to limit the maximal number of simultaneous connections in the server. The second requires the N-Server to be configured to generate code that queries the length of multiple queues. Each queue stores events of certain types. If there is a queue whose length exceeds its specified high watermark, then new connection requests are postponed, until the length drops below a specified low watermark. The second mechanism is effective in handling overload situations that can be caused by multiple bottlenecks, such as CPU and Disk [24].

The above illustrated that there is no single "right way" to build a network server. The N-Server must be configured based on the type of traffic expected, quality of service, fault tolerance, etc. Other supported features include performance profiling, debugging, logging, and termination of long-idle connections. Important statistical information of the server application can be automatically gathered, if the N-Server is configured to enable the performance profiling. This information includes: the number of connections accepted, the number of bytes read, the number of bytes sent, and the file cache hit rate, etc. The N-Server can generate network server applications in two modes: debug and runtime. If the server is generated in debug mode, then all internal events that are triggered in the server are written into a file. The user can trace this file to get a snapshot of what happened during the time an error condition occurred. Such a debug mechanism is far from being perfect, but it is useful. The N-Server can also be configured to generate network server applications with a logging capability. Long-idle connections may consume unnecessary resources and degrade the performance of network server applications. The N-Server can generate code that is able to automatically terminate these connections.

5.3 Applicability

The N-Server is applicable for the construction of server applications that use TCP Sockets as the communication mechanism. The generated server can act as both a server and a client, and it is also able to host multiple services, with each service registered on an individual server port.

5.4 Template Options

There exist many template options in the N-Server, which adapt many aspects of the template to the user's network server application. 13 of these options are lexical options. They serve two purposes: (1) avoid name conflicts and (2) let the user name and refer to the components in the generated framework. The second purpose is especially important to the N-Server, because it contains a large number of components and many of these components can be reused to implement functionalities in the user's network server

application. For example, the cache component is used in the N-Server for the automatic caching of disk files in memory; however, it can also be used to cache other objects. In CO_2P_3S, the value of a lexical option not only specifies the name of one component, but also affects the names of a set of derived components. For example, the class name of the component that realizes the LRU cache replacement policy is derived from the class name of the Cache component. There is a lexical option that specifies the class name of the Cache component, with default value "Cache". If the value "Cache" is changed to "XXXX", the derived class name of the LRU replacement policy is changed from "LRUCache" to "LRUXXXX". These are the lexical options:

- **Event Name**: The name of the base class that all the event classes should extend directly or indirectly. Its default value is "Event".
- **Component Name**: This is the name of the base class of all the Component classes. Its default value is "Component".
- **Event Handle Name**: The class name of the Event Handle class, which is the base class of all the Event Handler classes. Its default value is "Handle".
- **Event Handler Name**: The name of the interface that all the Event Handler classes should implement directly or indirectly. Its default value is "EventHandler".
- **Reactor Name** The name of the class that is responsible for the registration and deregistration of Event Handlers. Its default value is "Reactor".
- **Source Name**: This is the name of the base class that all the Event Source classes should extend directly or indirectly. Its default value is "Source".
- **Server Configure Name**: The name of the class that holds the configuration information of a Server Component. Its default value is "Server- Configuration".
- **Client Configure Name**: The name of the class that holds the configuration information of a Client Component. Its default value is "ClientConfiguration".
- **Linked List Name**: The name of the class that implements a fast linked list. Its default value is "LinkedList".
- **Queue Name**: The name of the class that implements a fast queue. Its default value is "Queue".

- **Buffer Element Name**: The name of the class that manages a memory buffer in the form of a byte array. Its default value is "BufferElement".
- **Cache Name**: The name of the Cache class, which is the base class of all the Cache classes. Its default value is "Cache".
- **Server Name**: The name of the Main class of the concurrent server application. It is compulsory for the user to specify its value. There is no default name.

Besides, there are in total twelve design options and performance options. Here, we only describe the possible selections of these options and defer most of the discussion of their meanings and effects to Section 5.5. These options are:

- **Reactor Dispatching Mode** *(Design Option)*: It can be set as either *Single* or *Multiple*, while the default value is Single. This option specifies whether the generated network server application uses a single Event Dispatcher or multiple Event Dispatchers.
- **Event Handling Mode** *(Design Option)*: It can be set as either *DISPACHER_THREADS* or *EVENT_PROCESSOR*, while the latter is the default value. This option specifies whether to use the Event Dispatcher thread or an Event Processor to process ready events.
- **Use Decoder Encoder** *(Design Option)*: It can be set as either *ON* (enabled) or *OFF* (disabled), while the former is the default value. If it's set to be *OFF*, the number of steps used to process a request is reduced to three. They are Read Request, Handle Request, and Send Reply (shown in Figure 5.2).

Figure 5.2: No decoder encoder

- **Use Completion Events** *(Design Option)*: It can be set as either *ON* or *OFF*, and its default value is *OFF*. If it is set to be *ON*, the type of the event processor that

is used to process completion events can be specified as either *SINGLE_CHANNEL_THREADS_POOL* or *MULTI_CHANNEL_THREADS_POOL*, while the default is *SINGLE_CHANNEL_THREADS_POOL*.

- **Event Processor Adjustable** *(Design Option)*: This option specifies whether to dynamically adjust the number of threads in an Event Processor. It takes effect if the Use Completion Events option is set to be *ON* or the Event Handling Mode is set to be *EVENT_PROCESSOR*. It can be set as either *ON* or *OFF*, and its default value is *OFF*.
- **Use File Caching** *(Design Option)*: This option takes effect if the Use Completion Events option is set to be *ON*. It can be set as either *ON* or *OFF*, and its default value is *OFF*. If it is set to be *ON*, the N-Server generates code to cache disk files in memory, and further, the cache replace policy can be specified as one of the following six values: *LRU, LFU, LRU-MIN, LRU-THRESHOLD, HYPER-G*, or *USER-DEFINED*.
- **Shutdown Long Idle Connections** *(Design Option)*: It controls whether the N-Server should generate code to enable the automatic shutdown of long-idle connections. This option can be set as either *ON* or *OFF*, and its default value is *OFF*.
- **Use Event Scheduling** *(Design Option)*: It controls whether to generate the necessary code underlying the Event Scheduling mechanism provided by the N-Server. This option can be set as either *ON* or *OFF*, and its default value is *OFF*.
- **Overload Control** *(Design Option)*: This option takes effect if the Use Completion Events option is set to be *ON* or the Event Handling Mode is set to be *EVENT_*PROCESSOR. It can be set as either *ON* or *OFF*. Its default value is *OFF*; in this case, the N-Server does not generate code that enables automatic overload control.
- **Application Mode** *(Performance Option)*: It specifies whether a network server application should be generated in debug mode. This option can be set as either *RUNTIME* or *DEBUG*. The former is the default value.

- **Performance Profiling Mode** (*Performance Option*): It can be set as either *ON* or *OFF*. This option specifies whether the N-Server should generate code to enable performance profiling. The default value of this option is *OFF*.
- **Logging Mode** (*Performance Option*): It can be set as either *ON* or *OFF*. The latter is the default value, which prevents the N-Server from generating code to support the logging capability.

5.5 Architecture of Generated Code

This section describes the architecture of the code generated by the N-Server. Four important aspects of the architecture are emphasized, which are the following: what are the participants, why are the participants required, how do the participants interact, and how does the setting of template options affect the structure of the template. A not implemented Logging server is used to illustrate these aspects.

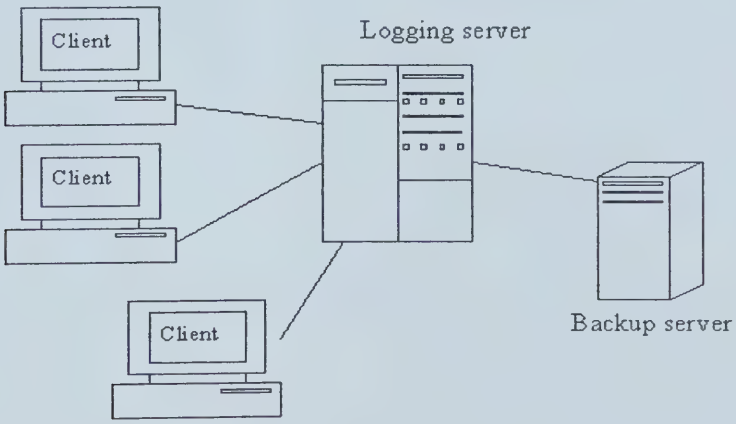

Figure 5.3: A centralized logging service

The Logging server provides a centralized logging service (shown in Figure 5.3) to network clients, which send logging requests to the Logging server. A logging request contains logging information in the form of a log record. The Logging server processes a logging request by writing the log record to a log file and then sends the status of the corresponding logging operation to the client. Unlike the request-response paradigm, a

client does not have to wait for a response from the server before sending the next request. However, the logging service must satisfy an order constraint: log records from the same client must be written to the log file in the order they are sent. After the log file reaches a certain size limit, the Logging server sends the log file to a Backup server for archiving.

Event	Purpose
Application Shut Down Event	Shut down the concurrent server application.
Client Open Event	Signal that an active connection is established.
Client Shut Down Event	Signal that an active connection is closed.
Compute Request Event	Represent a request from the peer.
Compute Result Event	Represent the result of handling a request.
Data Event	Data read from the peer.
End Communication Event	Signal the end of a connection.
# File Close Event	Close a file.
# File Delete Event	Delete a file or directory.
# File Open Event	Open a file. Create one if the file doesn't exist.
# File Read Directory Event	Read a file directory.
# File Read Event	Read from a file.
# File Write Event	Write to a file.
File Event State	The result of executing a file event.
Server Open Event	Signal the start of a server.
Server Shut Down Event	Signal the shut down of a server.
Socket Accept Event	Socket ready for new connections.
Socket Connect Event	Socket connected to the peer.
Socket Read Event	Data ready to read in the socket.
Socket Write Event	Socket ready to be written.
Start Communication Event	Signal the establishment of a connection.
Timer Event	A timer event.

Table 5.1: List of Events

The N-Server has the following key participants:

- **Event**: Embodies an event. It is implemented by a class, whose name is specified by the lexical option — Event Name. The Event class contains an *integer* field that stores the type of the event. This is to avoid the expensive type checking operations in Java. Another field in the Event class specifies from which handle the event is initiated. An Event can also be prioritized, if the Use Event Scheduling design option is set to *ON*. In this case, a private field called *priority* and two public methods that get and set the Event priority are added into the Event class. A list of Events is provided by the N-Server. This list is summarized in Table 5.1. Those Events whose name is prefixed by "#" are Completion Events.
- **Component**: Allows the state (any object) of the event handling to be attached to it and retrieved later on by the Event Handler. Event-driven concurrency models are relatively harder to program than process-per-connection or thread-per-connection models. The reason lies in the difference of how the two kinds of concurrency models manage the state of the required processing. In the latter models, the request processing state is saved or restored automatically by the OS during the process/thread switching time. However, in the former models, the user must explicitly manage the state, unless the processing of one request only concerns one single event. For example, decoding a logging request may need to process several Data Events, with each event the result of handling a Socket Read Event. Thus, the Component in the N-Server provides the necessary facilities for the managing of event handling states. The Component is implemented by a class, whose name is specified by the lexical option — Component Name. Five classes extend the Component class and they are: the Handle, the CommunicatorComponent, the ServerComponent, the ClientComponent, and the Container Component.
- **Handle**: Corresponds to the Handle in the Reactor design pattern. It's implemented by a class, whose name is specified by the lexical option — Handle Name. Three types of Handles are provided in the N-Server. They are Socket Handle, Timer Handle, and Queued Event Handle. A Socket Handle manages resources related to a socket. A Timer Handle manages resources of a timer. A

Queued Event Handle contains a virtual queue. Once an event is deposited into the queue, the Event Handler that is registered for this type of event is dispatched for execution. If an Event Handler is handling an event initiated from the Handle, it has state *busy*. Events initiated from the same Handle won't be dispatched for processing until the *busy* state is cleared. This gives rise to two benefits: (1) events initiated from the same Handle are handled in the order they arise, and (2) an Event Handler serializes the handling of events. Take the Logging service as an example. Although there may be multiple logging requests from the same client waiting in a Queued Event Handle, these requests are handled sequentially in the order they arrive. A Handle can also be *disabled or enabled*. If it is *disabled*, the Event Dispatcher stops to poll it for ready events. A programmer can disable a Handle to wait for a certain condition to occur and then reenable it to resume handling events initiated from the Handle. For example, the Logging server can postpone the handling of another logging request, until it completes writing the log record of the preceding logging request from the same client to the log file. This also helps to meet the ordering constraint of the logging service.

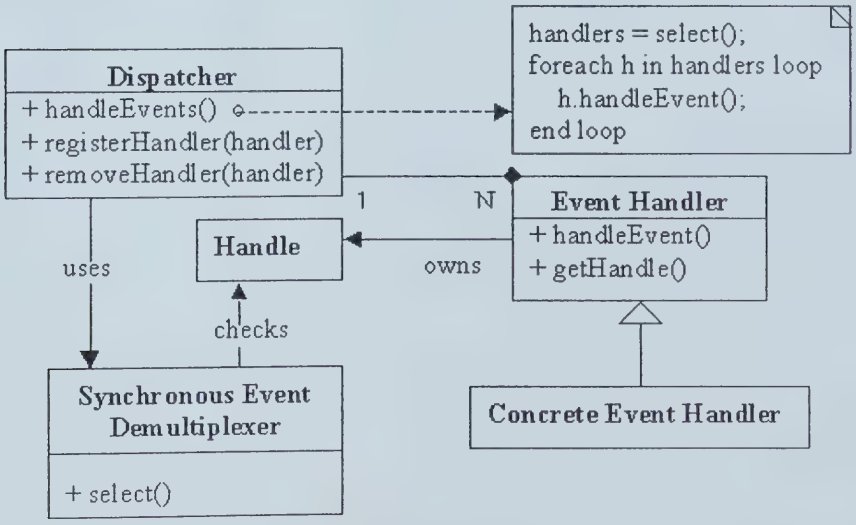

Figure 5.4: Event, Handle, and Event Handler

- **Event Handler**: Corresponds to the Event Handler in the Reactor design pattern. All Event Handlers implement an interface, whose name is specified by the lexical option — Event Handler Name. The interface contains a hook method that abstractly represents the operation for handling specific events. In the original Reactor design pattern, each Event Handler contained a Handle as a private field, so a new Event Handler instance had to be created each time it is needed. The N-Server avoids such performance and memory overheads by allowing an Event Handler to be completely stateless. This is possible, since the state of event handling can be stored and retrieved in the Handle object. Only a single instance of a stateless Event Handler needs to be created, because multiple Handles can share it. In addition, unlike the original Reactor design pattern, a Handle is registered with the Event Dispatcher, rather than an Event Handler. The relationship of the Event, the Handle, and the Event Handler is illustrated in a UML[13] class diagram (Figure 5.4).

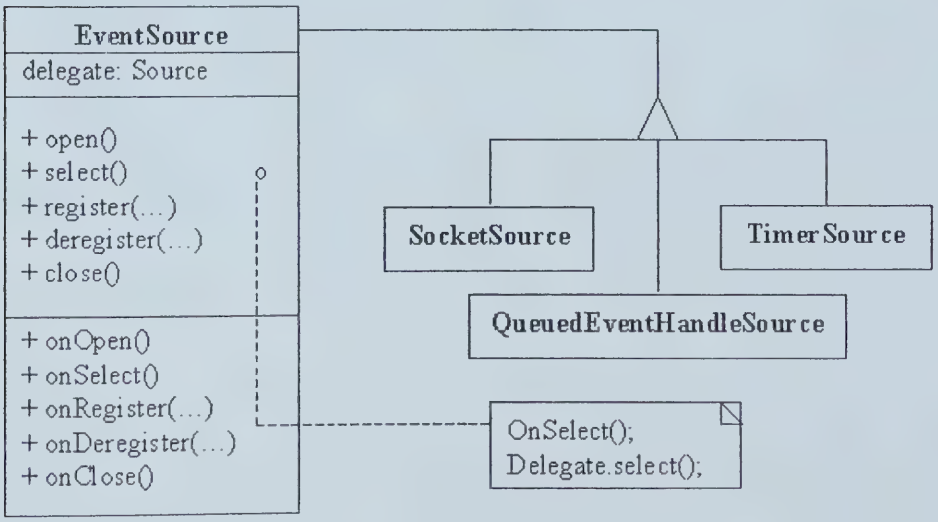

Figure 5.5: Event Sources

[13] UML stands for Unified Modeling Language.

- **Event Source**: Is the base class of all the Event Sources. Its name is specified by the lexical option — Source Name. An Event Source is responsible for the registration and deregistration of Handles, and polling of ready events. It complies with the Decorator design pattern [2], which decouples the management of different types of Event Select Sources and facilitates the addition of new Event Sources. The UML class diagram of the Event Source is shown in Figure 5.5. Three types of Event Sources are provided by the N-Server. They are the Socket Source, the Timer Source, and the Queued Event Handle Source. The Socket Source polls registered Socket Handles to check whether they are acceptable,[14] readable, writable, or connected. The *poll* operation is implemented using Java NIO. The Timer Source polls registered Timer Handlers for timer events. The Queued Event Handle Source polls which registered Queued Event handles have events queued onto them. The Queued Event Handle Source and the Queued Event Handle collaboratively deal with the handling of Completion Events or any user-defined events.
- **Communicator Component**: Plays the communication role with the network peer. Each Communicator Component contains a Socket Handle corresponding to an established socket connection. Two Event Handlers, the Read Request Event Handler and Send Reply Event Handler, are associated with this Socket Handle. It also contains three Queued Event Handles, which are the Decode Request Handle, the Compute Request Handle, and the Encode Reply Handle. If the Use Decoder Encoder design option is set to be *OFF*, the Decode Event Handle and Encode Event Handle no longer exist. This is illustrated in Figure 5.2. For brevity, the special case is not elaborated. When generating the Logging server, the Use Decoder Encoder design option is set to be *ON*, and three user-defined Event Handlers are supplied, which perform three functionalities respectively: decoding a logging request, handling a logging request, and encoding a logging reply. Let these three Event Handlers be called LogDecoder, LogComputer, and LogEncoder. The Decode Request Handle waits for Data Events passed by the

[14] A Socket Handle is acceptable, if there are pending connection requests. It is readable if there is data in the receive-buffer of its socket and writable if there is space in the send-buffer of its socket. It's connected when the connection operation to the remote peer is completed.

Read Request Event Handler and dispatches the LogDecoder to decode a logging request. If a logging request is decoded, a Compute Request Event that contains the decoded logging request is passed to the Compute Handle. The Compute Handle then dispatches the LogComputer to handle the logging request. The LogComputer writes the log record to the log file and passes a Compute Result Event to the Encode Reply Handle. The Compute result event contains the status of the logging operation. The Encode Reply Handle dispatches the LogEncoder to process the Compute Result Event and creates a Data Event that represents the encoded reply. Eventually, the Send Reply Event Handler sends the Data Event to the client. The Compute Request Event Handler is also dispatched to process the Start Communication Event and End Communication Event. The Start Communication Event is created when a connection is established with the client. This allows a network server generated by the N-Server to allocate necessary resources for processing requests or initiate communication with the client. The End Communication Event signals the server that a connection is terminated due to exceptional network conditions, so that the server can revoke resources allocated to the connection and do other necessary cleanups.

- **Read Request Event Handler**: Reads data from the Socket Handle with which it is associated and passes the Data Event to the Decode Request Event Handler of the Communicator Component.
- **Send Reply Event Handler**: Writes data to the remote peer. If the connection of the Communicator Component is terminated, it passes a Communicator Shut Down Event to the Queued Event Handle in the Server Component or a Client Shut Down Event to the Queued Event Handle in the Container Component.
- **Decode Request Event Handler**: Triggers the user-provided Event Handler to decode a request and passes a Compute Request Event to the Compute Handle of the Communicator Component.
- **Encode Reply Event Handler**: Triggers the user-provided Event Handler to encode a reply and creates a Data Event that is to be sent to the peer.
- **Compute Request Event Handler**: Triggers the user-provided Event Handler to handle Compute Request Events, Start Communication Events, and End

Communication Events. It passes a Compute Result Event to the Encode Reply Event Handle of the Communicator Component.

- **Event Processor**: Executes tasks using a pool of threads. Two types of Event Processors are provided in the N-Server, which are the Multi-Channel-Threads-Pool Processor and the Single-Channel-Threads-Pool Processor. The Single-Channel-Thread-Pool Processor is composed of a channel that holds the tasks to be executed and a pool of threads that repeatedly retrieves tasks from the channel. The task retrieval operation must be synchronized. This can cause contention if the granularity of the tasks is too small. To avoid over-contention over the channel, each thread retrieves a batch of tasks per retrieval, and the batch value can be set at runtime. The Multi-Channel-Threads-Pool Processor is composed of multiple Single-Channel-Threads-Pool Processors, and each Single-Channel-Threads-Pool Processor handles events of specific types.

  If the Event Processor Adjustable design option is set to be *ON*, a component called a Processor Controller is used to dynamically adjust the number of threads in a Single-Channel-Threads-Pool Processor. Such a capability helps to determine a good thread pool size, while meeting the concurrency requirements needed by events of certain types. It is often hard to determine the number of threads to be used in a Single-Channel-Threads-Pool Processor. For example, the maximal number of concurrent File I/O operations that can be handled by a normal UNIX machine is around 40 to 50 per second [19]. If too few threads are allocated, the disk IO capability provided by the OS is not fully utilized. However, allocating too many threads can also hurt the performance, due to the increased thread switching and synchronization overheads. The Processor Controller is similar to the Thread Pool Controller in SEDA [10]. It periodically monitors the number of tasks in the Channel of a Single-Channel-Threads-Pool Processor. If the number of tasks exceeds a certain threshold, the number of threads is increased by one until the maximal number of threads is reached. The Processor Controller also remembers the best throughput achieved by a specific thread pool size. If

allocating more threads doesn't improve the total throughput of the processor, it declines to do so.

The Channel in the Single-Channel-Threads-Pool Processor contains a synchronized queue. If the Event Scheduling design option is set to be *ON*, then the Channel becomes a set of prioritized queues, with each queue used to store tasks of a specific priority. Threads in the Single-Channel-Threads-Pool Processor poll the higher priority queues first. To avoid starvation of tasks of lower priority, after a designated number of tasks from a priority queue are retrieved, then the queue with the next highest priority is polled for tasks.

The Single-Channel-Threads-Pool Processor also plays a key role in supporting overload control. If the Overload Control design option is set to be *ON*, two fields, a low watermark and a high watermark, are added into the Single-Channel-Threads-Pool Processor class. If the number of tasks in its Channel is above the high watermark, then the server is considered overloaded, until the number of tasks drops below the low watermark.

- **Completion Event**: Refers to any event that can block the execution of a thread. All Completion Events should extend the base class: Completion Event. A Completion Event contains a Queued Event Handle. The execution result of the Completion Event is also an Event, which is deposited in the Queued Event Handle and later on dispatched to be handled by the Event Handler associated with the Queued Event Handler. The way a Completion Event is handled resembles that of the Completion Token design pattern [9], from which the Completion Event derives its name. At present, all the Completion Events provided by the N-Server are File Events. If the Use File Cache design option is set to be *ON*, the File Read Event queries the file cache first before issuing a file read operation.
- **Completion Event Processor**: Refers to the Event Processor that is used to handle Completion Events. It is not used if the Use Completion Events design option is set to be *OFF*. The type of the Event Processor is determined by the

design option — Use Completion Events. The Logging server uses the Completion Event Processor to simulate the existence of asynchronous Disk I/O.

- **Event Dispatcher**: It consists of a thread that repeatedly polls the Event Source associated with it for events. The ready events are processed directly by this thread, if the Event Handling Mode design option is set to be *DISPACHER_THREADS*. If the option is set to be *EVENT_ PROCESSOR*, the ready events are dispatched to an Event Processor called Reactive Event Processor for processing. The Reactive Event Processor is a Single-Channel-Threads-Pool Processor. If the Overload Control is set to be *ON*, the Event Dispatcher will poll all Event Processors to see whether they are overloaded. If so, it postpones the handling of the Socket Accept Event, which notifies the arrival of new socket connection requests from the network peer. The Logging server uses the Event Dispatcher to process ready events, since the Logging server is unlikely to handle heavy workloads.
- **Cache**: Provides the automatic caching of disk files in memory. It's implemented by a class, whose name is specified by the lexical option — Cache Name. This class can also be instantiated to cache other objects as well. The cache replacement policy is determined by the Use File Cache design option. If the cache replacement policy is set to be *USER-DEFINE*, the user must implement the *purge* hook method that replaces cache entries for space. The Logging server does not need the file caching capability.
- **Reactor**: It is responsible for delegating the registration and deregistration of Event Handles to the Event Dispatcher. It passes Completion Events to the Completion Event Processor, if the Use Completion Event design option is set to be *ON*. It contains the Reactive Event Processor, if the Event Handling Mode design option is set to be *EVENT_ PROCESSOR*. There can be one or more Event Dispatchers associated with it, depending on the value of the Reactor Dispatching Mode design option. The Reactor Log and the Performance Profiler are also associated with it, if the Logging Mode and the Performance Profiling mode are set to be *ON* respectively. The Reactor is implemented by a class, whose name is specified by the lexical option — Reactor Name. The Reactor class complies with

the Singleton design pattern. Based on different settings of three design options (the Event Handling Mode, the Reactor Dispatching Mode, and the Use Completion Events), the N-Server can realize eight different event-driven concurrency models (summarized in Table 5.2).

	Reactor Dispatching Mode	Event Handling Mode	Use Completion Events
I	Single	DISPATCHER_THREADS	OFF
II	Single	DISPATCHER_THREADS	ON
III	Single	EVENT_PROCESSORS	OFF
IV	Single	EVENT_PROCESSORS	ON
V	Multiple	DISPATCHER_THREADS	OFF
VI	Multiple	DISPATCHER_THREADS	ON
VII	Multiple	EVENT_PROCESSORS	OFF
VIII	Multiple	EVENT_PROCESSORS	ON

Table 5.2: Eight concurrency models

Model I is similar to the single-process-event-driven model (SPED), in that a single thread repeatedly executes the code that polls for ready events and then processes them. Model V enhances Model I by using multiple Event Dispatchers. Each Dispatcher has its own thread of control. Using multiple Event Dispatchers boosts the application performance and scalability by making use of multiple processors. Model II is similar to the multi-process-event-driven concurrency model (MPED), in that the Completion Event Processor can play the same role as the multiple *helper* processes/threads in MPED. Model VI enhances Model II in the same way as Model V does to Model I. Model III uses the Reactive Event Processor to handle ready events. Threads in the Reactive Event processor can be blocked when processing Completion Events. Model IV avoids this problem by using a Completion Event Processor. Model VII and Model VIII enhances Model III and Model IV respectively by using multiple Event Dispatchers. Multiple Event Dispatchers can make network servers designed to handle extremely high workloads more responsive, because the polling of ready events are carried out in parallel and ready events take less time to be dispatched.

- **Server Component**: Plays a passive connection role. It contains a Socket Handle, which listens to a user-specified network port for new connections. The Acceptor Event Handler is associated the Socket Handle. Whenever there is a new

connection request from a client, the acceptor Event Handler is dispatched to establish this connection. A Server Component also contains a Queued Event Handle with which the Server Event Handler is associated. Whenever a communication is accepted or terminated, a Start Communication Event or End Communication Event is sent to the Queued Event Handle, and the Server Event Handler is dispatched to do the corresponding administration tasks. Multiple Server Component instances can be created in the N-Server. This allows one network server to host multiple network services. The Logging server only needs one Server Component instance.

- **Acceptor Event Handler**: Associates with the Server Component to handle Socket Accept Events. It instantiates and initiates a new Communicator Component that holds the accepted connection. A Start Communication Event is sent to the Compute Handle of the Communicator Component.
- **Server Event Handler**: Handles Start Communication Events and End Communication Events. It is also triggered periodically to terminate long-idle passive connections, if the Shut Down Long Idle Connection design option is set to be *ON*.
- **Client Component**: Plays an active connection role. It contains one Communicator Component that corresponds to the connection with the network peer. With the Client Component, a network server generated by the N-Server can act as both a server and a client. The Logging server instantiates a Client Component to connect to the Backup server.
- **Connector Event Handler**: Associates with the Client Component to handle the establishment of an active connection with the remote peer. It instantiates and initiates a new Communicator Component that holds the accepted connection. A Start Communication Event is sent to the Compute Handle of the Communicator Component. The Acceptor Event Handler and the Connector Event Handler implements the idea of the Acceptor-Connector design pattern [8]. Therefore, although the N-Server only supports TCP Sockets currently, it can be easily augmented to support other communication mechanisms.

- **Container Component**: Serves as a compositor of a collection of Server and Client Components. It contains an Event Handle that waits for Server Open Events, Server Shut Down Events, Client Open Events, and Client Shut Down Events. The Application Event Handler is associated with this Handle and is dispatched to do the necessary administration tasks.
- **Application Event Handler**: Handles Server Open Event, Server Close Event, Client Open Event, and Client Close Event.
- **Client Configuration**: It is used to configure a Client Component. It is implemented by a class, whose name is specified by the lexical option — Client Configure Name. It specifies the IP address of the network peer and a list of socket options, such as the receive-buffer size, the send buffer size, whether to enable or disable the Nagle algorithm [31][15]. These socket options are used to configure the socket connection of the Client Component. The Logging server uses it to specify the IP address of the Backup server in order to archive a log file.
- **Server Configuration**: It is used to configure a Server Component. It is implemented by a class, whose name is specified by the lexical option — Server Configure Name. It specifies the port number the server listens to for new connections. Like the Client Configuration, it also specifies a list of socket options that connections from a server to clients should use. The Logging server uses it to specify the network port on which the logging service is provided.
- **Server**: It is the main class of the concurrent server application. The class name of the Server is specified by the lexical option — Server Name. Several hook methods used to configure a network server are included in this component.

The architecture of generated code varies greatly, based on different selections of design options and performance options. Table 5.3 summarizes how each option's selection affects the generated code of each component. All the options bring about changes that crosscut multiple components of the code generated by the N-Server. Some options (like

[15] The Nagle algorithm is aimed to reduce the number of small packets on WAN. The algorithm states that if a given connection has data sent but unacknowledged, then no small packets will be sent on the connection until the existing data is acknowledged.

Application Mode, Performance Profiling Mode, and Logging Mode) make structural changes that are so small and widespread that it is hard to describe them in detail.

	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII
Event				+				+				
Completion Event				O								
File Open Event				O		+						
File Read Event				O		+						
Handle	+											
File Handle				O		+						
Read Request Event Handler							+			+	+	+
Send Reply Event Handler							+			+	+	+
Decode Request Event Handler			O				+	+		+		+
Encode Reply Event Handler			O				+	+		+		+
Compute Request Event Handler			+	+			+	+		+		+
Event Processor					+			+	+	+		
Processor Controller					O							
Event Dispatcher		+		+					+	+	+	
Cache						O					+	
Reactor	+	+		+	+	+		+	+	+	+	+
Communicator Component			+				+	+			+	
Server Component			+				+			+		+
Client Component			+				+			+		+
Server Event Handler							+			+	+	
Connector Event Handler			+							+	+	+
Acceptor Event Handler			+						+	+	+	+
Container Component							+			+	+	+
Application Event Handler							+			+	+	
Client Configuration			+							+		
Server Configuration										+		
Server			+									

Table 5.3: Option selections vs. architecture changes

I: Reactor Dispatching Mode II: Event Handling Mode III: Use Decoder Encoder IV: Use Completion Events V: Event Processor Adjustable VI: Use File Cache VII: Shutdown Long Idle Connections VIII: Use Event Scheduling IX: Overload Control X: Application Mode XI: Performance Profiling Mode XII: Logging Mode

+: The option changes the component. O: The option decides the existence of the component.

The goal of the N-Server is to support the construction of a diverse range of network server applications, varying greatly from each other in terms of their functionality and performance requirements. It may be possible to achieve the same goal using a static object-oriented framework. However, we could expect a diffusion of indirection code in that case, which would inevitably degrade both the maintainability and the performance of the constructed network server applications.

5.6 Example

This section demonstrates the construction of a concurrent server application with the N-Server. A Time Server is constructed, which performs a simple function: it sends a time stamp to the client and then terminates the connection. The construction of this Time server has the following three steps:

1. Select the N-Server in CO_2P_3S.

```
import java.util.*;
public class TimeEventHandler implements EventHandler{
  public Event[] handleEvent(Event event){
    int type = event.getEventType(); // Event type
    if(type == Event.START_COMMUNICATION){
      StartCommunicationEvent startEvent = (StartCommunicationEvent)event;
      // Get the Communicator Component
      CommunicatorComponent comm = startEvent.getCommunicator();
      // Compute the current time and store it in a Data Event
      Date date = new Date();
      BufferElement bufferElement = new BufferElement(date.getString().getBytes());
      DataEvent dataEvent = new DataEvent(bufferElement, comm);
      // End the communication immediately.
      EndCommunicationEvent endEvent = new EndCommunicaitonEvent(comm,-1);
      Event events = new Event[] {dataEvent, endEvent};
      return events;
    }
    return null;
  }
}
```

Figure 5.6: Handle a time request

2. Configure the N-Server by specifying option values. Set the Server Name lexical option to be "TimeServer", which is also the main class name of the application. Use default values for the rest of the lexical options. Set the Reactor Dispatching Mode design option to be *Single*, the Event Handling Mode design option to be *DISPATCHER_THREADS*, and the Use Decoder Encoder design option to be *OFF*. Use the default value for the rest of design options and all the performance options. Then generate the framework of the Time Server.
3. Create an Event Handler class that is used to handle requests from the client. The Java code for this Event Handle is shown in Figure 5.6. Implement the hook method *getComputeEventHandle* in the TimeServer by adding one line: *return new TimeEventHandler();* implement the Hook method *getPort* in the TimeServer by adding one line: *return 9999*. This specifies that the Time server runs on port 9999.

The Time server uses a single thread to serve multiple simultaneous requests from clients. Less than twenty lines of non-commenting source code in total are needed to implement it.

Chapter 6

Applications

This chapter presents the design of two applications: COPS-HTTP, a high-performance HTTP server and COPS-FTP, an FTP server. Both of them are realistic applications of moderate size and generated using the Network Server design pattern template (N-Server). They are also the first HTTP server and FTP server created through an automatic code-generation approach. These two applications were developed to evaluate two different aspects of the N-Server: performance and flexibility. While COPS-HTTP was developed to verify the N-Server's ability to generate high performance network server applications[16], COPS-FTP was developed solely to demonstrate the flexibility of the N-Server to generate network server applications with complex architectures. This chapter described both the design of COPS-HTTP and the design of COPS-FTP to demonstrate the effectiveness of the N-Server in reducing the development complexity and development time of network server applications.

6.1 COPS-HTTP

Web servers form a key component of today's Internet services. The design and development of Web servers has become a difficult task, since they must handle many challenges imposed by the ever-growing popularity of the World Wide Web, including achieving high performance, improving service availability, and maintaining robustness. The existence of these challenges makes a web server a good candidate for demonstrating the N-Server's capability of easing development complexity, reducing development time, and achieving high application performance and scalability. Therefore, COPS-HTTP was developed using the N-Server.

The design objective of COPS-HTTP is to achieve high throughput and robust performance. Rather than being a full-featured Web server, COPS-HTTP handles only

[16] The performance evaluation of COPS-HTTP is presented in Chapter 7.

static Web page requests. It uses a single Event Dispatcher to dispatch ready Reactive Events to a Reactive Event Processor for processing. A Single-Channel-Threads-Pool Completion Event Processor is used to provide nonblocking disk I/O. COPS-HTTP is generated with the caching capability and enforced LRU cache replacement policy provided.

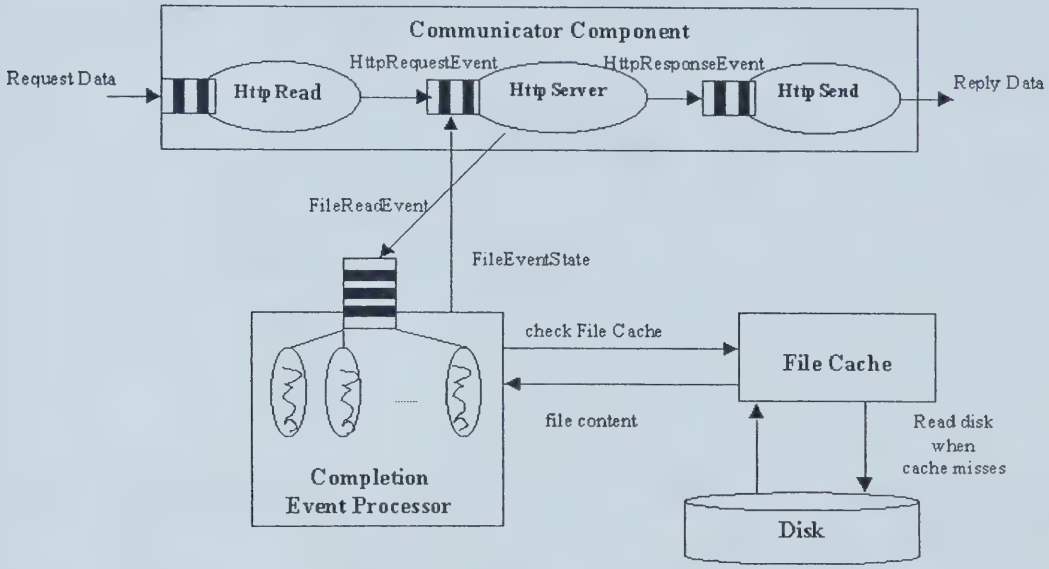

Figure 6.1: COPS-HTTP architecture

Figure 6.1 shows the structure of COPS-HTTP. Three Event Handlers are created: HttpRead, HttpServer, and HttpSend. The HttpRead parses an HTTP request contained in one or many Data Events to an HttpRequestEvent, and passes it to the HttpServer. The HttpServer reads the content of a requested Web page by sending a File Read Event to the Completion Event Processor, and then enters into a sleeping state[17] waiting the completion of the File Read Event. A File Read Event is handled by first checking whether it can be satisfied directly from the File Cache. Only when a file cache miss occurs is a blocking file read operation issued to read a file from disk. After the file read

[17] A sleeping state here does not mean that the thread processing an HttpRequestEvent is blocked. On the contrary, it turns to process other ready events.

operation is completed, a File Event State is created. It keeps the result and context of the file read operation and wakes up the HttpServer. The HttpServer resumes processing by creating an HttpResponseEvent that contains the appropriate HTTP response to the HttpSend. The HttpSend encodes an HTTP response into one or many Data Events and passes them to the SocketReadEventHandler (not shown in Figure 6.1), which eventually sends the response to a Web client.

	Classes	Methods	NCSS
Generated code	79	474	2,697
Application logic	16	89	785
Protocol library	10	50	449
Total	105	613	3,931

Table 6.1: The code distribution of COPS-HTTP

The code base of COPS-HTTP is composed of two parts: automatically generated code and handcrafted code. The handcrafted code can be further divided into code that constitutes an HTTP protocol library and code that implements the HTTP server-specific logic. Table 6.1 shows the code distribution. In total, there are 3,931 lines of non-comment source statements (NCSS) in COPS-HTTP, of which 2,697 lines of NCSS are automatically generated by the N-Server. If an existing HTTP protocol library were used for the development of COPS-HTTP, only 785 lines of NCSS would need to be programmed, which accounts for 20% of the total code of COPS-HTTP.

6.2 COPS-FTP

The nature of FTP has made an FTP server relatively harder to implement using the N-Server than a Web server. In FTP, a client establishes a control connection to a server actively. This control connection carries commands that tell the server which service to provide. However, unlike HTTP, FTP uses a separate data transfer connection for the file transfer [40]. A server can establish this data transfer connection either passively or actively. While HTTP is a stateless protocol that follows a simple request/response

paradigm, the FTP protocol requires state. An FTP server needs to maintain state information for each client connection, including a client's identity, data type, data structure, and transmission mode [38]. Besides, the context of some FTP requests must be kept, because several commands need to be exchanged between a server and a client to handle them. For example, to rename a file, a client first sends a *RENAME FROM* command that specifies the original pathname of the file. A server receives the command and then checks whether the file actually exists. If the file exists, it notifies the client to continue. Afterwards, the client sends a *RENAME TO* command that specifies the new pathname of a file.

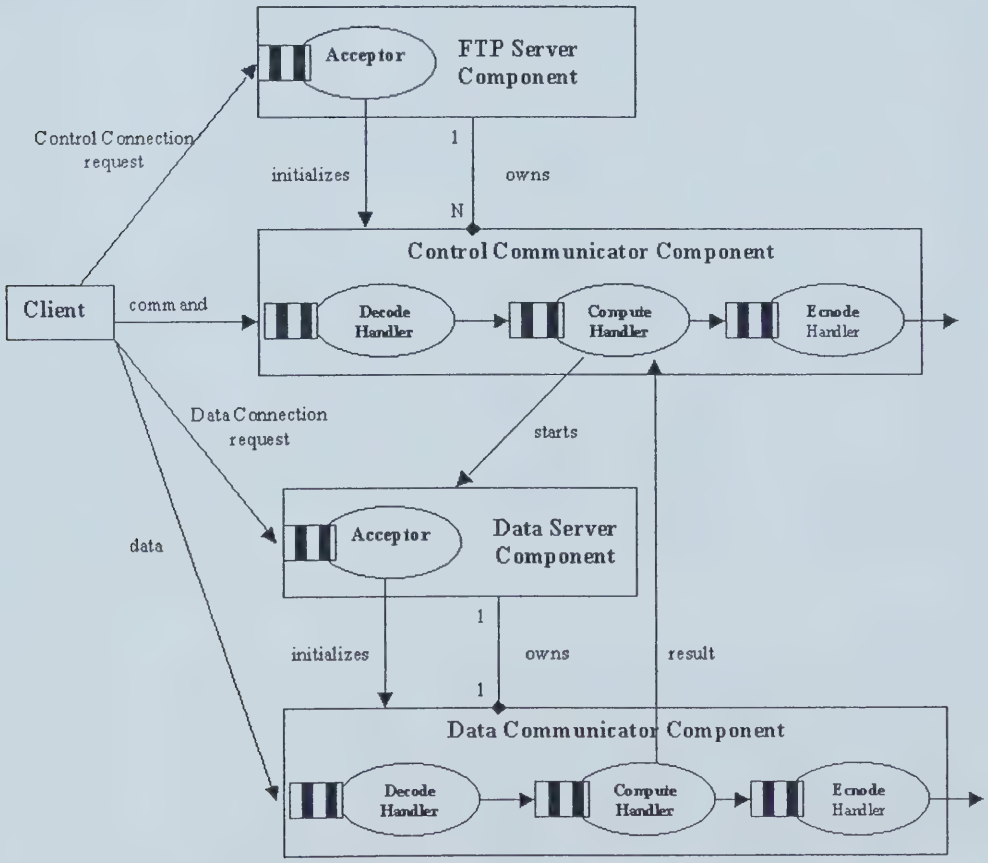

Figure 6.2: Storing a file

The complexity of FTP makes an event-driven FTP server more difficult to design and develop. Therefore, COPS-FTP was developed to demonstrate the flexibility of the N-Server.

COPS-FTP is a full-featured FTP server. Rather than building it from scratch like COPS-HTTP, we modified the Apache FTPServer, a Java-based multithreaded FTP server that is part of the Apache Avalon project [37]. COPS-FTP uses the same concurrency model as COPS-HTTP. Like COPS-HTTP, it also uses three event handlers that decode FTP commands, execute commands, and encode replies respectively. The peculiarity of COPS-FTP lies in how it performs file transfer over data transfer connections. For example, COPS-FTP can passively open a data transfer connection to store a file transferred from a client. The structure and interaction among several components involved in this example is shown in Figure 6.2. Upon receiving a *STORE* command that tells the server to receive a file, the server creates a new Data Server Component that listens to a port specified by the client. If the server needs to establish an active data transfer connection, a Client Component can be used, instead of a Server Component. When the data transfer connection arrives, a Communicator Component is initiated. Afterwards, the Communicator Component carries out the task of receiving a file and storing it on disk. When the file transfer completes, an event is created to notify the Compute Event Handler of the control Communicator Component. Eventually, both the Server Component and the Communicator Component used for the file transfer are removed and garbage collected.

	Classes	Methods	NCSS
Reused code	124	945	8,141
Removed code	18	199	1,186
Added code	23	150	1,897
Generated code	84	480	2,937

Table 6.2: The code distribution of COPS-FTP

Table 6.2 summarizes the code distribution of COPS-HTTP. A large fraction of code (8,141 lines of NCSS) from Apache FTPServer is reused, which supports many functions, including a GUI, a database or LDAP access, and user activity monitoring. A total of 1,897 lines of NCSS are added, replacing 1,186 lines of NCSS. 711 lines of extra NCSS have to be programmed, to migrate Apache FTPServer to an event-driven FTP server. Because of Java's built-in support for concurrency, only a few lines of code are needed to program multithreading. However, an event-driven application requires more code to deal with events explicitly. This gives rise to a slightly higher programming overhead and an increase of total application code size. Translating synchronous I/O operations to asynchronous I/O operations also incurs programming overhead. A synchronous operation may need only one line of code to make an I/O function call, while an asynchronous I/O operation takes several extra lines of code to save and restore the application state and context. Although this extra programming effort is needed, it is quite small compared to the number of NCSS generated automatically from the N-Server. Also, we believe COPS-FTP can achieve better performance than Apache FTPServer, because its event-driven concurrency model makes it capable of handling a large number of simultaneous connections without incurring the performance overheads caused by the presence of a large number of threads. This capability is of more importance to an FTP server than a Web server, since FTP connections are mainly long-lived [41].

6.3 Summary

The two cases studies described in this chapter reflect the following:

- An event-driven network server application is essentially more complex to develop than those using traditional concurrency models based on multiprogramming.
- The N-Server can greatly ease the cost of network server application development. With the N-Server, the major development cost of developing a network server application rests with the complexity of the application's network communication protocol and functionalities.
- The N-Server is fairly flexible and can be used to generate network server applications with relatively complex structures.

Chapter 7

Performance Evaluation

This chapter evaluates the performance of network server applications generated by the Network Server design pattern template (N-Server). Three experiments are conducted. The first experiment compares the performance of COPS-HTTP against the popular Apache Web server [32]. It is shown that COPS-HTTP not only outperforms Apache but also maintains a high degree of service fairness. The second experiment and third experiment demonstrate the effectiveness of the event scheduling and automatic overload control mechanisms provided by the N-Server. Only a minimal amount of programming effort is needed to incorporate these mechanisms in COPS-HTTP to achieve good quality of service (QoS).

7.1 The Performance Experiment

The N-Server can greatly ease the task of network server application development. However, it would not be of much help, if it could not generate network server applications that demand very high performance. This experiment was conducted to evaluate the performance of N-Server applications. It measures the performance of COPS-HTTP and Apache (version 1.3.27) under workloads of the form "get me a file". While COPS-HTTP is written completely in Java with most of the code generated using CO_2P_3S, Apache is handcrafted in C.

Apache is the most widely used web server on the Internet. A latest survey shows that more than 60% of web sites run Apache [38]. Apache implements the process-per-connection concurrency model and uses a pool of worker processes to serve simultaneous client connections. The size of the worker process pool is not static but adjusted dynamically based on the number of client connections. There is an upper bound on the number of worker processes. In this experiment, this bound is set to be 150. This figure was chosen since it is recommended by Apache's configuration documentation [32] and

can generally achieve the best performance [43]. Apache does not benefit from using more processes, due to the increasing overheads associated with process scheduling and switching.

The SPECweb99 is a commercial benchmark for evaluating the performance of World Wide Web servers [13]. The workload model of the SPECweb99 is composed of a static portion and a dynamic portion. The dynamic portion measures many dynamic functionalities of a Web server, such as Dynamic POST, Dynamic GET, CGI scripts, and cookies. The static portion models a hypothetical Web provider that hosts static Web pages from multiple clients. The file sizes and access frequencies of these pages are selected based on the traffic of several popular web sites. The file sizes range from 0.1 K to 1 MB and the average file size is 16 KB. The file access frequencies follow the Zipf distribution, with larger files accessed less frequently. The static portion accounts for 70% of the SPECweb99 load mix.

In this experiment, a set of files that represents the Web pages hosted by the Web provider is automatically created using a tool contained in the SPECweb99 benchmark suite. The total size of these files is 204.8 MB. The file cache size for COPS-HTTP is set to be only 20 MB, and therefore, a large amount of disk I/O needs to be done.

The performance metric of the SPECweb99 measures the maximal number of concurrent connections a web server can support, with the throughput of each connection not lower than a threshold. The workload generator contained in the SPECweb99 benchmark suite is designed to support such a performance metric, and it generates both static and dynamic workloads. However, our experiment is designed to measure the throughput of the two web servers under only static workloads. Therefore, a different workload generator[18] is used. This workload generator creates multiple threads on a client machine, with each thread simulating a Web client. Each Web client repeatedly performs the following actions: (1) establish a connection to the Web server, (2) send multiple requests for static Web pages according to the access distribution specified by the SPECweb99

[18] This workload generator is shipped together with the Haboob Web server.

benchmark, and (3) terminate the connection. The reason that the second action sends multiple requests is to simulate HTTP 1.1 persistent connections. After receiving a reply, a Web client pauses for 20 milliseconds before sending out the next request. This is to simulate the wide area network latency. Notably, the best-case round trip time (RTT) across the US is around 70 milliseconds [33]. The number of requests sent by a web client in the second action is 5. This value complies with the HTTP traffic pattern [33]. Before the benchmark measurements, both COPS-HTTP and Apache are warmed up. Each benchmark measurement is run for 5 minutes.

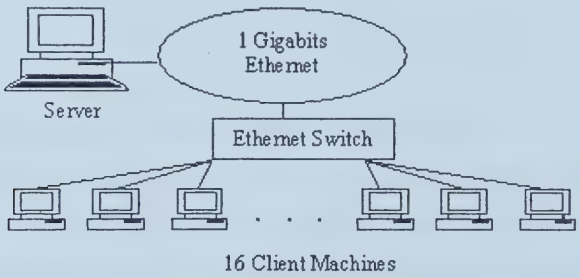

Figure 7.1: Experiment environment

The experiment environment is shown in Figure 7.1. Both web servers run on a Sun Enterprise 420R server with 4 450-MHz Sparc9 processors, 4 GB of RAM, and SunOS 5.9. 16 Sun Ultra 10 machines are used as clients for workload generation, with each machine having a 440-MHz UltraSparc processor, 256 MB of RAM, and SunOS 5.8. These client machines interconnect with the server over a switched Gigabit Ethernet. The maximal packet size of the Ethernet Switch is 1500 bytes. This complies with the SPECweb99 benchmark rules; however, the actual network bandwidth is limited to something slightly higher than 100 MBits/sec. Although the network bandwidth is clearly the bottleneck resource, this network configuration is quite close to the real-world situation that a high-performance Web server often faces.

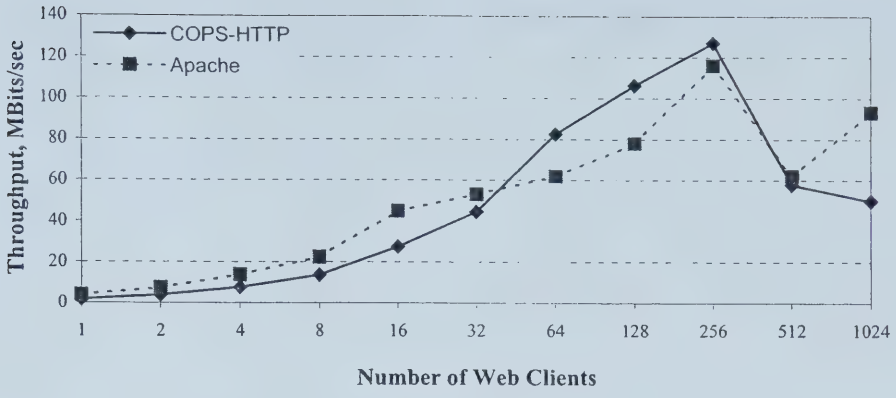

Figure 7.2: Web server throughputs

The number of Web clients simulated in the experiment varies from 1 to 1024. Figure 7.2 shows the throughput of both COPS-HTTP and Apache. Apache achieves slightly better throughput than COPS-HTTP when there are less than 32 web clients. However, this does not imply that COPS-HTTP is inferior to Apache under light workloads. The reason for the slightly poorer throughput is actually based on a lower response time of for Apache. Under light workload, the average response time[19] of Apache is smaller than the COPS-HTTP. The relatively longer average response time of COPS-HTTP results from queuing delays inherent in almost all the event-driven concurrency models. Even with one single Web client, it still takes COPS-HTTP around 30 milliseconds to handle a request, while the average response time of Apache in such a case is almost negligible. The client workload generator translates this slower response into fewer requests, since the faster a web server responds, the more requests the client workload generator makes. Under light workloads, more requests are generated by a fixed number of clients for Apache than COPS-HTTP, because Apache has a smaller average response time. This results in lower throughput for COPS-HTTP. A different workload generator that does not wait for responses before generating new requests could produce different throughput statistics. Fortunately, although COPS-HTTP has relatively longer average response time

[19] The response time is measured from the time a request is sent to the time a reply is received by a web client.

than Apache under light workloads, this response time is too small to be noticed by a web client, and hence, causes no service dissatisfaction.

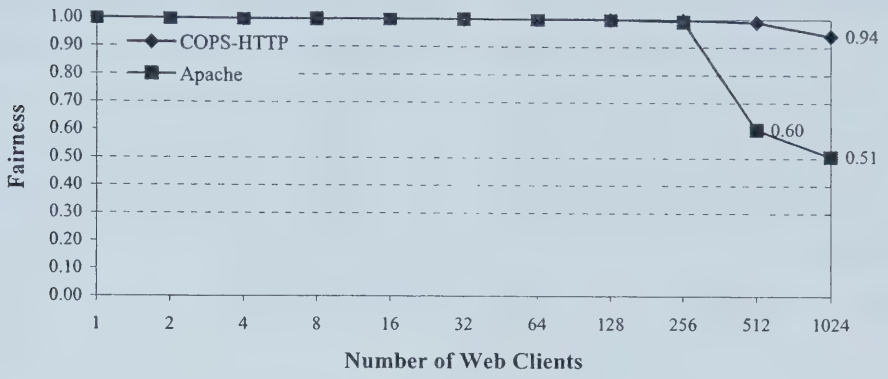

Figure 7.3: Service fairness

When the number of web clients is between 32 and 256, the throughput of COPS-HTTP is higher than that of Apache. This result confirms that the advantage of event-driven concurrency models lies in its superior performance and scalability to handle a large number of simultaneous requests. With more than 256 web clients, both COPS-HTTP and Apache get saturated, because the network becomes the performance bottleneck. In this case, the throughput of Apache drops less than that of COPS-HTTP, at the expense of extreme service unfairness.

Fairness is an important performance metric that is concerned with the equal allocation of resources. Figure 7.3 shows the service fairness metric that is based on the Jain fairness index [34] of the number of responses received by each Web client. This metric is computed from $f(x) = \left(\sum x_i\right)^2 / N\sum x_i^2$, where x_i is the number of responses received by Web client *i*, and N is the total number of Web clients. If all the Web clients receive an equal number of responses, the fairness index is 1. If *k* Web clients receive equal number of responses, and the other Web clients receive no responses at all, the fairness index is *k/N*. Under heavy loads, the fairness index of COPS-HTTP remains high, while Apache's fairness index drops significantly. With 1024 Web clients, the fairness index of Apache is

merely 0.51. The extreme unfairness of Apache is caused by the exponential backoff scheme of the TCP protocol. Apache only handles 150 simultaneous connections at any time. For a lucky web client, its connection is accepted, and a single process handles all its requests quickly. Unfortunately, some web clients are very unlucky, in that their TCP SYN packets for establishing connections are dropped by Apache. In this case, they may wait for a significant amount of time before doing a retransmit. The maximal retransmission timeout under Solaris is 1 minute. With a large number of clients receiving little service or no service at all, Apache better utilizes the limited network bandwidth to handle more requests from those lucky clients. Therefore, Apache achieves higher throughput than COPS-HTTP under very heavy workloads, at the expense of fairness.

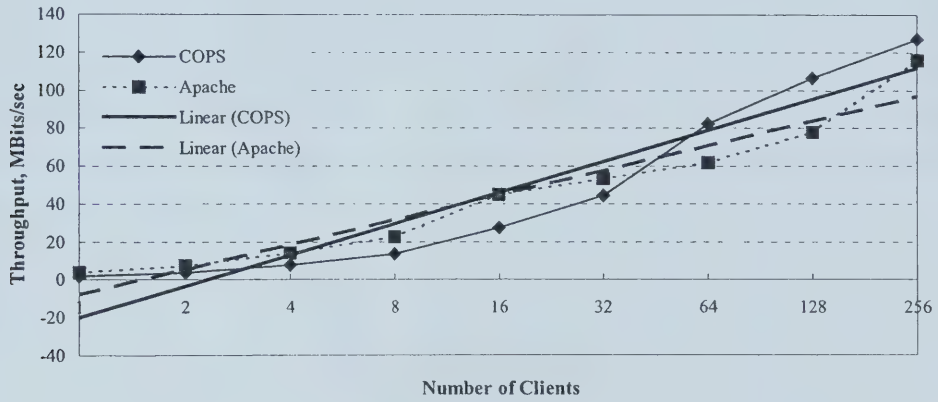

Figure 7.4: Throughput linear models

Figure 7.4 shows the throughput trends of two Web servers based on their linear models. The linear model for Apache is $y = 13.119 \bullet \log_2(x) - 7.832$, while the one for COPS-HTTP is $y = 16.487 \bullet \log_2(x) - 19.838$.

Although the full performance advantage of COPS-HTTP over Apache is not revealed in the experiment under limited network bandwidth, we can still draw a conclusion that COPS-HTTP generally outperforms Apache while maintaining a high degree of service

fairness. Accordingly, the N-Server is suited for the generation of network server applications with high performance demand.

7.2 The Event Scheduling Experiment

Providing different levels of QoS is an important issue to many network server applications. This experiment demonstrates how this issue can be addressed by the N-Server with its event scheduling mechanism, and accordingly verifies the effectiveness of this mechanism.

One scenario is used, which hypothesizes an Internet Service Provider (ISP) providing hosting services to two types of Web content, a corporate portal and personal homepages. The corporation has higher service requirements and can afford more service charges than individuals. Therefore, Web accesses to the corporate portal are prioritized by allocating more system resources (for example, bandwidth, disks, processors, memory, etc).

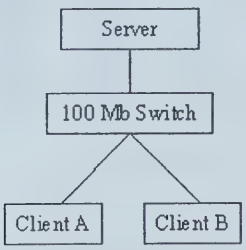

Figure 7.5: Event scheduling experiment environment

The COPS-HTTP Web server is used to provide the Web content hosting service. Two 933 MHz Pentium III systems with 256 MB of RAM and Linux 2.4.9 are used as client machines to generate workloads (shown in Figure 7.5). Rather than creating separate sets of files to represent the two types of contents, the single file set in the first experiment is used. To achieve the effect of double file sets, requests sent from one client machine are considered as accesses to the corporate portal, while requests from the other one are considered as accesses to personal homepages. The origin of a request can be determined

easily by checking the *from* IP address of the request. The N-Server is reconfigured to generate a variant of COPS-HTTP that incorporates an event scheduling mechanism. The file caching capability is disabled to make the workload heavier in terms of disk I/O. The event scheduling mechanism of the N-Server is fairly easy-to-use. Only 13 lines of code are added to the standard COPS-HTTP Web server in order to implement the functionalities required by this scenario.

The COPS-HTTP variant runs on a dual-processor 600 MHz Pentium III machine with 512 MB of RAM and Linux 2.4.9. The server is interconnected with the two client machines over a 100 MBits/sec Ethernet. Each client machine simulates 64 web clients. Other experiment parameter values are similar to those in the first experiment.

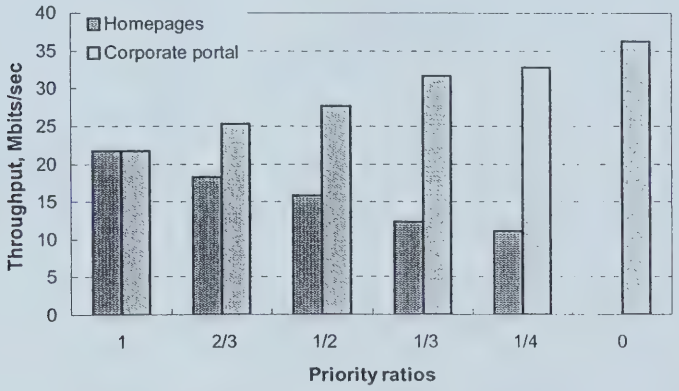

Figure 7.6: Corporate portals vs. homepages

Figure 7.6 shows the throughput of requests for the two types of contents under various priority level settings. A priority level setting is specified as a ratio x/y, where x gives the priority level of homepages and y specifies the priority of the corporate portal. A ratio x/y also mandates that a $x/(x+y)$ fraction of all kinds system resources be devoted to serve requests for homepages. The rightmost ratio on the x-axis shows the throughput of requests for the corporate portal, when the client machine that generates requests for homepages is off-line. There is a small gap between the ratio of priority levels and the

actual throughput ratio of requests for two types of Web contents. However, such a gap is quite acceptable, because the COPS-HTTP variant exerts no control over the management and scheduling of many operating system resources, such as the order in which the network socket buffers are drained.

In conclusion, this experiment demonstrates that the event scheduling mechanism of the N-Server is both effective and easy-to-use.

7.3 The Overload Control Experiment

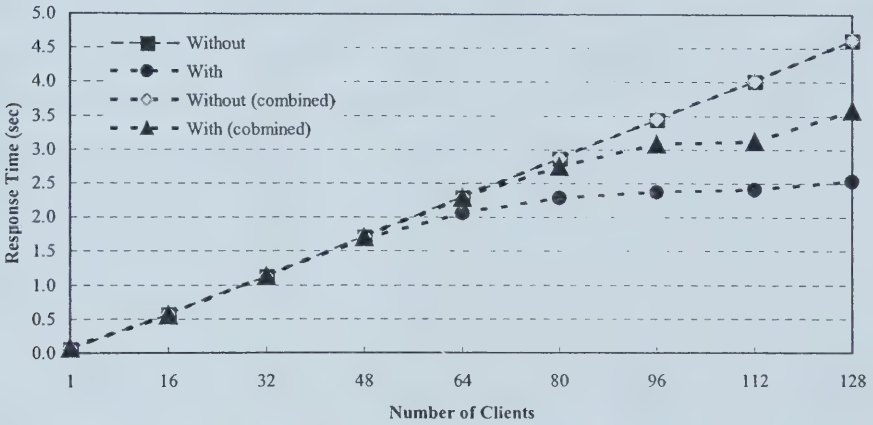

Figure 7.7: Response time

Among all the performance metrics of a Web server, the one that a Web client cares about the most is the response time [33]. This experiment demonstrates how the automatic overload control mechanism provided by the N-Server can be used to reduce the response time when a web server is overloaded, and therefore, improve the satisfaction of web clients. Two variants of the standard COPS-HTTP Web server are generated using the N-Server. One variant has the automatic overload control capability and the other does not. The automatic file caching capability is also disabled for both of them to increase workload. This experiment uses the same experimental setup as the event scheduling experiment. CPUs are considered as the bottleneck resource that causes server overload. Unfortunately, workloads generated by the client workload generator are I/O bound, rather than CPU-intensive. To overcome this problem, each thread is forced to

sleep for 50 milliseconds when decoding an HTTP request. The high watermark and low watermark for the Reactive Event Processor are set to 20 and 5 respectively. When there are more than 20 events waiting in the queue of the Reactive Event Processor, the COPS-HTTP variant that has automatic overload control capability stops accepting new connections until the number of waiting events drops below 5. The number of web clients in this experiment varies from 1 to 128.

Figure 7.7 shows the average response time of the two variants. The combined response time counts the time a web client waits to establish a connection when calculating the response time. The response time alone better describes what Web clients with established connections experience when a Web server gets overloaded. When overloaded, a server should concentrate on preventing the QoS of established connections from dropping rather than taking on more tasks. In this sense, the response time is more meaningful than the combined response time. However, the average combined response time is also presented, because it describes the experiences of all the web clients (including those with established connections and those without) as a whole.

Figure 7.7 shows that the automatic overload control mechanism significantly reduces both the average response time and the average combined response time. This achieved without degrading the server throughput.

This experiment does not reveal many capabilities of the automatic overload control mechanism provided by the N-Server, such as handling overload conditions caused by multiple bottleneck resources. However, it still successfully verifies the effectiveness of this mechanism.

7.4 Summary

The three experiments presented in this chapter demonstrate the following two properties of the N-Server:

- The N-Server is able to generate network server applications that support very high performance demand. In addition, it achieves high performance without sacrificing service fairness.
- The event scheduling and automatic overload control mechanisms provided by the N-Server are both effective and easy-to-use to achieve good QoS in the generated network server applications.

Chapter 8

Summary and Conclusions

8.1 Conclusions

CO_2P_3S is intended to greatly ease the complexity of software system development by using a pattern-based generative application development approach. This approach had been proved fairly effective for generating small parallel and sequential programs. Whether CO_2P_3S is suitable for generating larger, realistic applications is a more important criterion for evaluating the effectiveness of CO_2P_3S. Accordingly, this research applied CO_2P_3S to the construction of network server applications. As a result, the Network Server design pattern template (N-Server) was constructed. The N-Server contains a large number of template options, which make it fairly reconfigurable and suitable for the generation of network server applications with a diverse range of performance and functionality requirements. Two moderate-sized network server applications, an HTTP server (COPS-HTTP) and an FTP server (COPS-FTP), were generated using the N-Server. While the latter demonstrated the N-Server's capability of generating network server applications with complex structures, the former showed that the N-Server is suitable for generating network server applications with very high performance requirements. Experiences obtained through the construction of these two applications indicate that the N-Server can greatly ease the complexity of network server application development by generating a large fraction of application code. As to performance, an experiment was conducted to measure the performance of COPS-HTTP against the popular Apache Web server under static workloads. COPS-HTTP has comparable performance in general and achieves better throughput than Apache under high workloads. In addition, two other experiments demonstrated the effectiveness of the N-Server's event scheduling and automatic overload control mechanisms, which are two desirable functionalities needed by many network server applications to respond to different levels of quality-of-service. In conclusion, this research has successfully applied CO_2P_3S to the construction of network server applications and provided evidence of the

effectiveness of the pattern-based generative application development approach of CO_2P_3S.

8.2 Future Work

Currently, the N-Server only supports the generation of network server applications that use TCP for network communication. One potential area of future work is to enable the N-Server to support UDP. Although UDP lacks many of the key features provided by TCP (such as lost packet retransmission, duplicate detection, and congestion control), it is used in place of TCP in several circumstances. For instance, UDP is used rather than TCP when an application needs to use multicasting or broadcasting. UDP can also be used to achieve high-speed bulk-data transfer over reliable networks like LAN, since it has no cost associated with connection establishment and shutdown like TCP. An UDP-based network server application is often hard to design and develop, since it has to implement one or many missing features provided by TCP. The N-Server can be enhanced to deal with this problem. In the enhanced N-Server, the actual network communication protocol can be made into a design option. This option allows a developer to choose whether to use TCP or UDP as the underlying network communication protocol of the network server application to be generated. Further, a developer should be shielded from the low-level network communication details of UDP or TCP, and program against the same set of high-level events. Besides, the enhanced N-Server should also include a set of design options, which control whether to generate code that implements one or many missing features provided by TCP. We believe that the enhanced N-Server can significantly reduce the development complexity of UDP-based network server applications.

Another future project is to enhance the automatic overload control mechanism of the N-Server. The existing mechanism of the N-Server detects an overload condition by checking whether the number of events waiting for services exceeds a user-specified threshold. Although this mechanism is effective, it is relatively hard to use, because it requires a programmer to roughly estimate the average processing time of events. To solve this problem, a rate-based overload control mechanism can be added into the N-Server. In this new mechanism, the threshold is not fixed but adjusted dynamically, based

on a user-specified target event-handling rate of an Event Processor. If the actual event-handling rate is higher than the target rate, the threshold can be leveraged to allow the server to accommodate more client connections. Otherwise, the threshold should be lowered to make the server respond quickly to overload conditions.

Cohort scheduling is a technique that organizes the computation in server applications to improve performance [39]. Its key idea is to defer processing a request until a cohort of similar requests are accumulated and then process them in a batch manner. This improves instruction and data locality, since these requests have similar computations. In the future research, we can add the capability of cohort scheduling into the N-Server.

Finally, we can also extend the N-Server to generate distributed network server applications that execute in a distributed-memory environment consisting of a network of workstations (NOW). A NOW is not only much cheaper than a single large server of equivalent processing power but also more fault-tolerant. Previous research [20] has successfully applied CO_2P_3S to generate parallel programs executing in a NOW environment, resulting in new parallel design templates. These templates abstract a programmer from almost all the intricacies inherent in distributed-memory parallel programming, such as network communication, synchronization, and load balancing. Based on the experiences gained from this research, we believe that the pattern-based generative development approach should also be able to significantly ease the complexity of distributed network server application development.

Bibliography

[1] J. Hu and D.C Schmidt. JAWS: A Framework for High Performance Web Servers. In Domain-Specific Application Frameworks: Frameworks Experience by Industry, Wiley & Sons, 1999, pp. 339-376.

[2] R. Johnson, E. Gamma, R. Helm, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994, pp. 1-2.

[3] D.C. Schimdt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley-Sons, 2002.

[4] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, K. Tan. Generative Design Patterns. 17th IEEE International Conference on Automated Software Engineering (ASE), September 2002, Edinburgh, UK, pp. 23-34.

[5] J. Anvik. Asserting the Utility of CO_2P_3S Using the Cowichan Problems. Master's Thesis, Department of Computing Science, University of Alberta, Fall 2002.

[6] R. Hitchens. Java™ NIO. O'Reilly, 2002.

[7] T. Harrison, I. Pyrarli, D.C. Schmidt, and T. Jordan. Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers. Washington University technical report #WUCS-97-34, September 1997.

[8] D.C. Schmidt. Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Services. In Pattern Languages of Program Design, (R. Martin, F. Buschman, and D. Riehle), Reading, MA: Addison-Wesley, 1997.

[9] T.H. Harrison and D.C. Schmidt. Asynchronous Completion Tokens: An Object Behavioral Pattern for Efficient Asynchronous Event Handling. In Pattern Languages of

Program Design, (R. Martin, F. Buschman, and D. Riehle), Reading, MA: Addison-Wesley, 1997.

[10] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-01), ACM SIGOPS Operating Systems Review, Vol 35, 5, ACM Press, October 2001, pp. 230-243.

[11] Zeus Inc. The Zeus WWW Server. http://www.zeus.co.uk

[12] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In USENIX Annual Technical Conference (USENIX-99), Monterey, California, USA, June 1999, pp. 199-212.

[13] The Standard Performance Evaluation Corporation. SpecWeb96/SpecWeb99. http://www.spec.org/osg.

[14] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic code generation from design patterns. IBM Systems Journal, 35(2), 1996, pp. 151-171.

[15] A. Eden, J. Gil, Y. Hirshfeld, and A. Yehudai. Towards a Mathematical Foundation for Design Patterns. Technical Report, Department of Information Technology, Uppsala University, and Tel Aviv University, Number 1999-004, May 1999.

[16] G. Florijn, M. Meijers, P. van Winsen, Tool Support for Object-Oriented Patterns. In Aksit, M., Matsuoka S. (eds), Proceedings of ECOOP'97, Jyv'fiskyli, Finland, 1997. Lecture Notes in Computer Science I24I. Berlin: Springer Verlag, 1997, pp. 134-143.

[17] ModelMaker Tools. Design patterns in ModelMaker. http://www.modelmaker.demon.nl/mm_design_patterns.htm.

[18] S. MacDonald, D. Szafron, and J. Schaeffer, "Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks", in Proceedings of the 5th USENIX Conference on Object-OrientedTools and Systems (COOTS'99), San Diego, California, USA, May 1999, pp. 29-43.

[19] J.C. Hu, I. Pyarali, and D.C. Schmidt. High Performance Web Servers on Windows NT: Design and Performance. Proceedings of the USENIX Windows NT Workshop, August 11--13, 1997, Seattle, Washington, USENIX, 1997, pp. 149-149.

[20] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. to appear in PPoPP'2003: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[21] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E.A. Fox. Caching Proxies: Limitation and Potentials. Technical Report, Virginia Polytechnic Institute and State University, Number TR-95-12, July 18, 1995.

[22] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World Wide Web Documents. Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM SIGCOMM Computer Communication Review, Vol. 26,4, ACM Press, August 26-30 1996, pp. 293-305.

[23] T. Voigt. Overload Behaviour and Protection of Event-Driven Web Servers. Lecture Notes in Computer Science, Vol. 2376, pp. 147-157, 2002.

[24] T. Voigt, and P. Gunningburg. Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls. Lecture Notes in Computer Science, Vol. 2334, 2002, pp. 50-68.

[25] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 20--23, 2001, Schloss Elmau, Germany, IEEE Computer Society Press, 2001, pp. 139-146.

[26] J. Almedia, M. Dabu, A. Manikntty, Pei Cao. Providing differentiated levels of service in web content hosting. Technical Report, University of Wisconsin, Madison, Number CS-TR-1998-1364, March 1998.

[27] D.C. Schmidt, M. Stal, H. Rohnert and F. Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. John Wiley & Sons, 2000, pp. 14-14.

[28] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture - A System of Patterns. John Wiley & Sons, 1996, pp. 29-31.

[29] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchial Internet Object Cache. In Proceedings of the 1996 Usenix Annual Technical Conference, . Jan. 1996, pp. 153-163.

[30] Gnutella. http://gnutella.wego.com.

[31] R. Stevens. UNIX Network Programming. Volume 1, Second Edition: Networking APIs: Sockets and XTI. Prentice Hall, 1998, pp. 202-202.

[32] Apache Software Foundation. The Apache Web server. http://www.apache.org.

[33] J. C. Mogul. The case for persistent-connection HTTP. Computer Communication Review, 25(4), October 1995, pp 299--313.

[34] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research, Sep. 1994.

[35] D.C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, in Pattern Languages of Program Design, Reading, MA: Addison-Wesley, 1995, pp. 529-545.

[36] J. Postel, J. Reynolds. File Transfer Protocol. RFC 959. ISI, Oct. 1985.

[37] Apache Avalon Project. http://avalon.apache.org.

[38] August 2003 Web Server Survey. http://www.netcraft.com/survey.

[39] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. Proceeding of the Workshop on Optimization of Middleware and Distributed Systems (OM-01), *ACM SIGPLAN Notices*, Vol. 36, 8, ACM Press, June 18 2001, pp. 182-187.

[40] D. E. Comer, D. Stevens. TCP/IP Volumn I: Client-Server Programming and Application. Prentice Hall, 1997, pp. 419-426.

[41] K. Thompson, G. Miller and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. IEEE Network, Novermber, 1997, pp. 10-23.

[42] D. E. Comer, D. Stevens. TCP/IP Volumn III: Client-Server Programming and Application. Prentice Hall, 1997, pp. 177-188.

[43] M. Arlitt and C. Williamson. Understanding Web Server Configuration Issues. Software-Practice and Experience, 2000, pp. 1-7.

University of Alberta Library
0 1620 1829 9675
B45838